Integrated Lectures Notes on Logic

Florian Rabe

2008-2022

Contents

This is a collection of notes from my undergraduate and graduate lectures on logic.

Standard textbooks for logic are [**?**], [**?**], and [**?**] (look for the respective latest edition). [**?**] is an interesting textbook that emphasizes higher-order logic and proof theory.

Part I

First-Order Logic

Chapter 1

Syntax

1.1 Syntax of Propositional Logic

Definition 1.1 (PL-signatures)**.** A PL-signature is an alphabet, i.e., a set Σ of symbols.

Definition 1.2 (PL formulas)**.** The PL-grammar for the signature Σ is:

$$\begin{array}{lcll}
\texttt{form} & ::= & \mathit{true} & \mathit{truth} \\
 & \mid & \mathit{false} & \mathit{falsity} \\
 & \mid & \texttt{form} \wedge \texttt{form} & \mathit{conjunction} \\
 & \mid & \texttt{form} \vee \texttt{form} & \mathit{disjunction} \\
 & \mid & \texttt{form} \rightarrow \texttt{form} & \mathit{implication} \\
 & \mid & \neg\texttt{form} & \mathit{negation} \\
 & \mid & p \text{ where } p \in \Sigma & \mathit{boolean\ variables}
\end{array}$$

The set of words over this grammar is the set $\mathbf{Sen}^{PL}(\Sigma)$ of Σ-*formulas* (also called Σ-*propositions* or Σ-*sentences*).

This grammar is of course ambiguous. If we actually want to parse words, we have to make it unambiguous by adding brackets around conjunctions, disjunctions, and implications.

Remark 1.3. It is a specialty of propositional logic that all formulas are well-formed, i.e., the well-formed PL-formulas over Σ are a context-free language. Normally, the languages are context-sensitive.

1.2 Syntax of First-Order Logic

Definition 1.4 (FOL-Signatures)**.** A FOL-signature is a triple (Σ_p, Σ_f, ar) where Σ_p and Σ_f are disjoint sets of symbols and $ar : \Sigma_p \cup \Sigma_f \to \mathbb{N}$ is a mapping.

The symbols in Σ_p and Σ_f are called *predicate symbols* and *function symbols*, respectively. If $ar(s) = 0, 1, 2, 3, 4, \ldots$, we say that s is a *nullary*, *unary*, *binary*, *ternary*, 4-ary, ..., symbol.

Definition 1.5 (FOL-Grammar)**.** The FOL-grammar for the signature $\Sigma = (\Sigma_p, \Sigma_f, ar)$ is given by the productions in Fig. 1.1 with start symbol `form`.
The language of this grammar is denoted by $Form^{FOL}(\Sigma)$.

Words produced from `term` are called *terms*. Words produced from `form` are called *formulas* (or propositions or sentences).
Words produced from `var` are variable names. These are elements of some countable set that is usually left implicit. For example, we can put

$$L(\texttt{var}) = \{x_n, y_n, z_n \mid n \in \mathbb{N}\}.$$

`form`	$::=$	*true*	*truth*
	$\mid$	*false*	*falsity*
	$\mid$	`form` $\wedge$ `form`	*conjunction*
	$\mid$	`form` $\vee$ `form`	*disjunction*
	$\mid$	`form` $\rightarrow$ `form`	*implication*
	$\mid$	$\neg$`form`	*negation*
	$\mid$	$\forall$`var form`	*universal quantification*
	$\mid$	$\exists$`var form`	*existential quantification*
	$\mid$	$p(\underbrace{\texttt{term}, \ldots, \texttt{term}}_{n})$ where $p \in \Sigma_p$ with $ar(p) = n$	*atomic formulas* or *predicates*
`term`	$::=$	$f(\underbrace{\texttt{term}, \ldots, \texttt{term}}_{n})$ where $f \in \Sigma_f$ with $ar(f) = n$	
	$\mid$	`var`	
`var`	$::=$	some symbol	*variables*

Figure 1.1: FOL Grammar

This grammar is also ambiguous. To make it unambiguous, we have to add brackets around all complex formulas.

Remark 1.6. The intuition behind the arity is that it gives the number of arguments that a function or predicate symbol takes.

Arity 0 is permitted: Nullary function symbols are also called *constant* symbols, nullary predicate symbols are the *boolean variables* from propositional logic.

Remark 1.7. The intuition behind the non-terminals `term` and `form` is that terms represent mathematical objects and formulas represent properties of mathematical objects.

It is easy to see that the FOL syntax is an extension of the PL-syntax.

Note that formulas and terms are not mixed: In every branch of the syntax tree of a well-formed formula, all `form`-nodes occur above all `term`-nodes.

Equality is needed so often that we define FOLEQ. It is like FOL but with a fixed binary predicate symbol $\doteq$ that is always present.

Definition 1.8 (FOL with equality)**.** FOLEQ is defined like FOL but with one production added to the grammar: `form` $::=$ `term` $\doteq$ `term`.

Remark 1.9 (Equality)*.* It is crucial to understand the difference between $=$ and $\doteq$.

$\doteq$ is a symbol of the alphabet, which happens to look very similar to the symbol $=$. But at this point $\doteq$ has no meaning whatsoever. It just takes two term and turns them into a formula.

$=$ on the other hand, has a meaning: It denotes equality of mathematical objects. In particular, we can use $=$ to express the equality of words over our alphabets, e.g., the equality of terms and formulas. For example, we can say the following: "We assume $F = s \doteq t$." This sentence expresses that F is a word (in this case a formula) over the alphabet and F is the formula $s \doteq t$.

Example 1.10 (Set Theory)*.* Axiomatic set theory was introduced around the 1920s by Zermelo and Fraenkel. It is the base of all modern mathematics. In modern language and notation, a very simple variant of axiomatic set theory is based on the following FOLEQ-signature:

- a binary predicate symbol $\in$, and we usually write $s \in t$ instead of $\in(s, t)$
- a nullary function symbol $\varnothing$,
- a unary function symbol Pow,
- a unary function symbol $\bigcup$,

- a binary function symbol *unorderedpair*, and we usually write $\{s,t\}$ instead of $\mathit{unorderedpair}(s,t)$.

Example formulas for this signature are

- $\forall x\ \neg x \in \varnothing$,
- $\forall x\ \forall y\ x \in \{x,y\}$,
- $\forall x\ \forall y\ \{x,y\} \doteq \{y,x\}$,
- $\forall x\ \varnothing \doteq \{x,x\}$,
- $\forall x\ \forall y\ x \in y$.

Note the difference between the first three and the last two formulas: The first three feel "correct" or "true", whereas the last two feel "false". However, at this point, no formula has a meaning. They are all just words over our grammar.
This signature looks like it has only very few and very boring terms, for example $\varnothing$, $Pow(\varnothing)$, $\{\varnothing,\varnothing\}$, and so on. And the formulas are even more boring because there is only one predicate symbol. However, together with some abbreviations, this is already enough to talk about most of mathematics.
For example, mathematics usually uses the following abbreviations for terms and formulas:

- If s and t are terms, then the term $\{s,\{s,t\}\}$ is abbreviated as (s,t).
- If s and t are terms, then the term $\bigcup\{s,t\}$ is abbreviated as $s\cup t$ (or $(s\cup t)$ if there are ambiguities otherwise).
- If s is a term, then the term $\{s,s\}$ is abbreviated as $\{s\}$.
- If s and t are terms (which do not use the variable x), then the formula $\forall x\ (x \in s \rightarrow x \in t)$ is abbreviated as $s \subseteq t$.

Then we can give even more abbreviations:

- We write 0 instead of $\varnothing$.
- We write 1 instead of $\{\varnothing\}$.
- We write 2 instead of $\{\varnothing,\{\varnothing\}\}$.
- We write 3 instead of $2\cup\{2\}$.
- and so on

Example 1.11 (Monoids and Groups). Monoids and groups (as well as rings, fields, etc.) can also be expressed as FOLEQ signatures. The signature for monoids has:

- a binary function symbol $\circ$, and we usually write $s \circ t$ instead of $\circ(s,t)$ (again we may have to write $(s \circ t)$ to prevent ambiguities),
- a nullary function symbol e.

The signature for groups extends the signature for monoids with

- a unary function symbol *inv*, and we usually write s^{-1} instead of $\mathit{inv}(s)$.

1.3 Contexts and Substitutions

Free and Bound Variables Bottom-up induction on syntax trees is the most important induction for logics and similar languages. It is often just called "induction on the grammar", or "induction on the language", or even just "induction". The following definitions are done in this way. The following definitions are for FOLEQ; the analogues for FOL are obtained by deleting the case for $\doteq$.

Definition 1.12 (Bound Variables)**.** We define the set $BV(w)$ of *bound variables* of a word w — i.e., a formula or a term — as follows.

- for terms:
  - $BV(x) = \varnothing$,
  - $BV(f(t_1, \ldots, t_n)) = BV(t_1) \cup \ldots \cup BV(t_n)$,
- for formulas:
  - $BV(p(t_1, \ldots, t_n)) = BV(t_1) \cup \ldots \cup BV(t_n)$,
  - $BV(t_1 \doteq t_2) = BV(t_1) \cup BV(t_2)$,
  - $BV(\mathit{true}) = \varnothing$,
  - $BV(\mathit{false}) = \varnothing$,
  - $BV(F \wedge G) = BV(F) \cup BV(G)$ and accordingly for the other connectives,
  - $BV(\forall x\ F) = BV(F) \cup \{x\}$ and accordingly for the other quantifier.

The opposite of bound variables are the free variables.

Definition 1.13 (Free Variables)**.** We define the set $FV(w)$ of *free variables* of a word w — i.e., a formula or a term — as follows.

- for terms:
  - $FV(x) = \{x\}$,
  - $FV(f(t_1, \ldots, t_n)) = FV(t_1) \cup \ldots \cup FV(t_n)$,
- for formulas:
  - $FV(p(t_1, \ldots, t_n)) = FV(t_1) \cup \ldots \cup FV(t_n)$,
  - $FV(t_1 \doteq t_2)) = FV(t_1) \cup FV(t_2)$,
  - $FV(\mathit{true}) = \varnothing$,
  - $FV(\mathit{false}) = \varnothing$,
  - $FV(F \wedge G) = FV(F) \cup FV(G)$ and accordingly for the other connectives,
  - $FV(\forall x\ F) = FV(F) \setminus \{x\}$ and accordingly for the other quantifier.

Remark 1.14. Note that:

- $x \in BV(F)$ iff there is a $\forall x$ or a $\exists x$ in F. We say that, e.g., $\forall x\ F$ binds the variable x in F. $\forall$ and $\exists$ are called *binders*.
- For a term t, we always have $BV(t) = \varnothing$ because terms cannot contain binders. This is a special property of FOL and FOLEQ — for example, lambda calculus is a language where terms may contain binders (namely the λ binder).
- The sets $FV(F)$ and $BV(F)$ are always finite.
- The definitions of $BV(-)$ and $FV(-)$ only differ in two cases.
- If a variable x occurs in F, then $x \in BV(F)$ or $x \in FV(F)$ or both. For example, x occurs both free and bound in $p(x) \wedge \forall x\ q(x)$.

Remark 1.15. A Σ-formula/term with an empty set of free variables is called *closed* or a *ground formula/term*. Ground formulas are often called *sentences* and

If F is a formula with $FV(F) = \{x_1, \dots, x_n\}$, then $\forall x_1 \dots \forall x_n\ F$ and $\exists x_1 \dots \exists x_n\ F$ are closed formulas. We call them the *universal closure* and the *existential closure* of F.

Example 1.16. Here are some examples for free and bound variables:

- For the formula $F = x \circ (y \circ z) \doteq (x \circ y) \circ z$ over the signature of monoids, we have $BV(F) = \varnothing$ and $FV(F) = \{x, y, z\}$. Its universal closure is $\forall x\ \forall y\ \forall z\ x \circ (y \circ z) \doteq (x \circ y) \circ z$.
- There are not so many ground terms over the signature of monoids: They are e, $e \circ e$, $(e \circ e) \circ e$, $e \circ (e \circ e)$, and so on.
- If a FOL- or FOLEQ-signature has no constant symbols, then there are no ground terms.
- If a FOL-signature has a unary predicate symbol p, then $\exists x\ p(x)$ is a (non-boring) ground formula.
- If a FOLEQ-signature has a unary function symbol f, then $\exists x\ x \doteq f(x)$ is a (non-boring) ground formula.

The last two examples show that even with only one function or predicate symbol, FOL and FOLEQ already offer a lot of interesting formulas to study.

Contexts Above we defined the syntax FOLEQ by using a context-free grammar. This grammar produced all formulas, i.e., formulas with any number of free variables. This is often not desirable because we usually only care about closed formulas.

However, the language of closed formulas is not context-free. This is easy to see: Every binder $\forall x\ F$ declares a variable x; this variable is *local* in the sense that it has a certain *scope* and should only occur within its scope; for example, in $G \wedge \forall x\ F$, the scope of x is F (but not G). In general, all languages with such declarations are not context-free. Therefore, we can only give a context-free grammar for formulas but not for closed formulas.

This is the main reason why we use formulas with free variables at all: We can write down a context-free grammar for them, and such a grammar gives us the powerful and simple induction principle of induction on syntax trees.

In general, we use contexts to keep track of the free variables:

Definition 1.17 (Context)**.** Let Σ be a signature. A Σ-*context* Γ is a finite set of variables.

We define

$$\Gamma \vdash_\Sigma F : \mathtt{form} \qquad \text{iff} \qquad F \in \mathit{Form}^{FOLEQ}(\Sigma) \text{ and } FV(F) \subseteq \Gamma$$

If $\Gamma \vdash_\Sigma F : \mathtt{form}$, we say that F is a Σ-formula in context Γ. If $\Gamma = \varnothing$, we write $\vdash_\Sigma F : \mathtt{form}$.

Finally, we define

$$\mathbf{Sen}^{FOLEQ}(\Sigma) = \{F \mid \vdash_\Sigma F : \mathtt{form}\}$$

Example 1.18. Let Σ be the signature of monoids. The formula $x \circ (y \circ z) \doteq (x \circ y) \circ z$ is a formula in the context $\{x, y, z\}$:

$$x, y, z \vdash_\Sigma x \circ (y \circ z) \doteq (x \circ y) \circ z : \mathtt{form}$$

It is also a formula in any bigger context:

$$x, x', y, z, y' \vdash_\Sigma x \circ (y \circ z) \doteq (x \circ y) \circ z : \mathtt{form}$$

Substitutions We can think of free variables as holes or placeholder that can be filled with arbitrary terms. This filling operation is the task of substitutions.

Definition 1.19 (Substitution)**.** Let Σ be a signature. A *substitution* from a Σ-context Γ to a Σ-context Γ' is a mapping from Γ to Σ-terms in context Γ'.

Notation 1.20. If γ is a substitution from Γ to Γ', we write $\gamma : \Gamma \to \Gamma'$. If $\Gamma = \{x_1, \dots, x_n\}$ and $\gamma(x_i) = t_i$, we write γ as $[x_1/t_1, \dots, x_n/t_n]$.

Remark 1.21 (Contexts). It is not so common to use contexts when talking about FOL and FOLEQ. When you study additional literature, you will probably not find them. The reason is that — contrary to other logics — FOL and FOLEQ are so simple that contexts are not needed. However, contexts make a lot of things much clearer and more elegant, and a good computer scientist should understand them.

Notation 1.22. Some literature writes substitutions the other way around: t/x instead of x/t.

Example 1.23. Let Σ be the signature of monoids.

- If $\Gamma = \{x, y, z\}$ and $\Gamma' = \{x, u, v, y\}$, then $[x/y, y/x, z/e \circ v]$ is a substitution from Γ to Γ'.
- The identity $[x_1/x_1, \dots, x_n/x_n]$ is a substitution from the context $\{x_1, \dots, x_n\}$ to itself.
- More generally, $[x_1/x_1, \dots, x_n/x_n]$ is a substitution from the context $\{x_1, \dots, x_n\}$ to any context Γ' with $\{x_1, \dots, x_n\} \subseteq \Gamma'$.

The intuition behind a substitution $[x_1/t_1, \dots, x_n/t_n]$ from Γ to Γ' is that every (free) variable x_i occurring in a formula/term over Γ is substituted with t_i thus creating a formula/term over Γ'. This is formalized in the following definition.

Definition 1.24 (Substitution application). Let Σ be a signature and $\gamma : \Gamma \to \Gamma'$ a substitution between Σ-contexts. Then we define the *application* $\overline{\gamma}(-)$ of γ to a word as follows:

- terms:
  - $\overline{\gamma}(x) = \gamma(x)$,
  - $\overline{\gamma}(f(t_1, \dots, t_n)) = f\big(\overline{\gamma}(t_1), \dots, \overline{\gamma}(t_n)\big)$,
- formulas:
  - $\overline{\gamma}(p(t_1, \dots, t_n)) = p\big(\overline{\gamma}(t_1), \dots, \overline{\gamma}(t_n)\big)$,
  - $\overline{\gamma}(t_1 \doteq t_2) = \overline{\gamma}(t_1) \doteq \overline{\gamma}(t_2)$,
  - $\overline{\gamma}(\mathit{true}) = \mathit{true}$,
  - $\overline{\gamma}(\mathit{false}) = \mathit{false}$,
  - $\overline{\gamma}(F \wedge G) = \overline{\gamma}(F) \wedge \overline{\gamma}(G)$ and similarly for the other connectives,
  - $\overline{\gamma}(\forall x\ F) = \forall x\ \overline{\gamma^x}(F)$ and similarly for the other quantifier.

  Here γ^x is a substitution from $\Gamma \cup \{x\}$ to $\Gamma' \cup \{x\}$ defined by $\gamma^x(x) = x$ and $\gamma^x(y) = \gamma(y)$ for all variables y in Γ. (In the special case where $x \in \Gamma$, the former takes precedence.)

Notation 1.25. If F is a formula with $FV(F) = x_1, \dots, x_n$, it is common to use the following abbreviation:

$$F(t_1, \dots, t_n) \quad := \quad \overline{[x_1/t_1, \dots, x_n/t_n]}(F).$$

And similarly, if t is a term with $FV(t) = x_1, \dots, x_n$, then:

$$t(t_1, \dots, t_n) \quad := \quad \overline{[x_1/t_1, \dots, x_n/t_n]}(t).$$

Moreover, we use the following abbreviations

$$F[x_1/t_1, \dots, x_n/t_n] \quad := \quad \overline{[x_1/t_1, \dots, x_n/t_n]}(F).$$

$$t[x_1/t_1, \dots, x_n/t_n] \quad := \quad \overline{[x_1/t_1, \dots, x_n/t_n]}(t).$$

There is one tricky problem with substitutions. Assume $\gamma = [x/y] : \{x\} \to \{y\}$. Then $\overline{\gamma}(\forall y\ p(x,y)) = \forall y\ p(y,y)$. This is not intended: We want to obtain something like $\forall y'\ p(y,y')$, i.e., we want to distinguish the first y, which should stem from the context $\{y\}$ and the second one, which should stem from the binder $\forall y$. If a substitution makes two variables equal that are supposed to different, we speak of *variable capture*. In this example, we can avoid variable capture only by renaming y to y'. This is called *α-renaming*.

Definition 1.26 (α-renaming)**.** Replacing a (sub-)formula $\forall x\ F$ with $\forall x'\ F'$ where F' is like F but with all occurrences of x replaced with x' is called an *α-renaming* (from x to x'). The same holds for $\exists$ instead of $\forall$.

More generally, we can always avoid variable capture as follows: Before applying the substitution $\gamma : \Gamma \to \Gamma'$ to the formula F, we use α-renaming to replace all variables $x \in BV(F) \cap \Gamma'$ with other variables. If we want to indicate that we do that, we speak of *capture-avoiding substitutions*.

Example 1.27. Let Σ be the signature of monoids. Let $\Gamma = \{x, y, z\}$ and $\Gamma' = \{x, u, v, y\}$, and $\gamma = [x/y, y/x, z/e\circ v]$ as above. Then

- $\overline{\gamma}\big(x \circ (y \circ z) \doteq (x \circ y) \circ z\big) \quad = \quad y \circ (x \circ (e \circ v)) \doteq (y \circ x) \circ (e \circ v)$,
- $\overline{\gamma}\big(\forall z\ x \circ (y \circ z) \doteq (x \circ y) \circ z\big) \quad = \quad \forall z\ y \circ (x \circ z) \doteq (y \circ x) \circ z$,
- $\overline{\gamma}\big(\forall y\ x \circ (y \circ z) \doteq (x \circ y) \circ z\big) \quad = \quad \forall y\ y \circ (y \circ (e \circ v)) \doteq (y \circ y) \circ (e \circ v)$
  (variable capture)
- $\overline{\gamma}\big(\forall y'\ x \circ (y' \circ z) \doteq (x \circ y') \circ z\big) \quad = \quad \forall y'\ y \circ (y' \circ (e \circ v)) \doteq (y \circ y') \circ (e \circ v)$
  (capture-avoiding variant of the above after α-renaming of y to y')

Eventually we can state the main property of substitutions:

Theorem 1.28 (Closure under Substitution)**.** *Let Σ be a signature and $\gamma : \Gamma \to \Gamma'$ a substitution between Σ-contexts. Then:*

- *if* $\Gamma \vdash_\Sigma t : \texttt{term}$*, then* $\Gamma' \vdash_\Sigma \overline{\gamma}(t) : \texttt{term}$*,*
- *if* $\Gamma \vdash_\Sigma F : \texttt{form}$*, then* $\Gamma' \vdash_\Sigma \overline{\gamma}(F) : \texttt{form}$*,*

where, if necessary, the substitution application uses α-renaming in F such that variable capture is avoided.

Proof. This is left as an exercise. □

1.4 An Abstract Definition of the Syntax of a Logic

Above we have seen the signatures and formulas of three logics: PL, FOL, and FOLEQ. And there are many more logics: Researchers have developed all kinds of logics for different purposes. For example, two general ways how to obtain new logics from old ones are the following.

- Changing the signatures: for example, ALGEQ (algebraic equational logic) arises from FOLEQ by only using signatures without predicate symbols.
- Changing the formulas: for example, HORNEQ (equational Horn logic) arises from FOLEQ by only allowing formulas that look like this:
  $$\forall x_1 \ldots \forall x_m\ ((F_1 \wedge \ldots \wedge F_n) \to F)$$
  where all the $F_1, \ldots, F_n, F$ are atomic (i.e., they do not contain proper subformulas).

Therefore, it is important to study the abstract properties of logics so that we can work with all these logics at once. Otherwise, every definition or theorem would have to be repeated for every logic.

Therefore, we introduce the following definition.

Definition 1.29. A *logic syntax* is a pair $(\mathbf{Sig}, \mathbf{Sen})$ where $\mathbf{Sig}$ is a collection of objects — called the *signatures* — and $\mathbf{Sen}$ is a mapping from $\mathbf{Sig}$ to sets, i.e., if $\Sigma \in \mathbf{Sig}$, then $\mathbf{Sen}(\Sigma)$ is a set. The elements of $\mathbf{Sen}(\Sigma)$ are called the Σ-*sentences*.

Example 1.30. Now we can say:

- Let $\mathbf{Sig}^{PL}$ be the collection of PL-signatures, and $\mathbf{Sen}^{PL}$ be the mapping that maps a PL-signature to its set of PL-formulas. Then $(\mathbf{Sig}^{PL}, \mathbf{Sen}^{PL})$ is a logic syntax.
- Let $\mathbf{Sig}^{FOL}$ be the collection of FOL-signatures, and $\mathbf{Sen}^{FOL}$ be the mapping that maps a FOL-signature to its set of FOL-sentences. Then $(\mathbf{Sig}^{FOL}, \mathbf{Sen}^{FOL})$ is a logic syntax.
- The logic syntax $(\mathbf{Sig}^{FOLEQ}, \mathbf{Sen}^{FOLEQ})$ is defined similarly to $(\mathbf{Sig}^{FOL}, \mathbf{Sen}^{FOL})$.

When defining logics (or related formalisms) in this way, we typically end up with the following situation:

- There is a collection of signatures Σ.
- For every signature, there is a collection of contexts $Con(\Sigma)$.
- For every signature Σ and every context Γ, there is the set of formulas $\Gamma \vdash_\Sigma F : \mathtt{form}$.
- For a fixed signature Σ, we can use substitutions $\gamma : \Gamma \to \Gamma'$ between two Σ-contexts Γ and Γ'.
- Such a substitution $\gamma : \Gamma \to \Gamma'$ induces a mapping from formulas $\Gamma \vdash_\Sigma F : \mathtt{form}$ to formulas $\Gamma' \vdash_\Sigma \overline{\gamma}(F) : \mathtt{form}$.

For example, this is the case for FOL and FOLEQ. FOL and FOLEQ have the special property that for all signatures Σ, the contexts are simply finite sets of variables. In other logics, the contexts can be more complex.

1.5 Theories

The above definition of logic syntax comes in very handy when we define theories: Theories are defined in exactly the same way for every logic. With our definition of logic syntax, we can handle all logics at once.

Definition 1.31 (Theories)**.** Let $L = (\mathbf{Sig}, \mathbf{Sen})$ be a logic syntax. Then an L-*theory* is pair (Σ, Θ) where $\Sigma \in \mathbf{Sig}$ and $\Theta \subseteq \mathbf{Sen}(\Sigma)$. The formulas in Θ are called the *axioms* of the theory.

The most important logic, for which theories are used, is FOLEQ. A FOLEQ-theory is a pair of a FOLEQ signature and a set of FOLEQ-formulas. Let us look at some examples of FOLEQ-theories.

Example 1.32 (Set Theory). Below we will use the abbreviation $F \leftrightarrow G$ for $(F \to G) \wedge (G \to F)$. The theory of set theory consists of the above-mentioned signature for set theory and the following axioms:

- axiom of equality (also called extensionality, intuitively: Two sets are equal if they have the same elements.): $\forall X\, \forall Y\, (X \doteq Y \leftrightarrow \forall z\, (z \in X \leftrightarrow z \in Y))$,
- axiom characterizing the empty set: $\forall x\, \neg x \in \varnothing$,
- axioms characterizing the unordered pair: $\forall x\, \forall y\, \{x, y\} \doteq \{y, x\}$ and $\forall x\, \forall y\, x \in \{x, y\}$,
- axiom characterizing the union: $\forall X\, \forall z\, (z \in \bigcup X \leftrightarrow \exists x\, (x \in X \wedge z \in x))$,
- axiom characterizing the power set: $\forall X\, \forall z\, (z \in Pow(X) \leftrightarrow z \subseteq X)$,
- some other axioms, which would go beyond the scope of this lecture. A nice overview is given at http://en.wikipedia.org/wiki/Zermelo%E2%80%93Fraenkel_set_theory. However, there the variant of set theory is used where $\in$ is the only function/predicate symbol.

As said above, this formulation of set theory goes back to works by Zermelo and Fraenkel in the 1920s [?, ?]. However, it was not originally written in FOLEQ. In fact, FOLEQ was not even fully understood yet at the time. It has undergone various changes in modern mathematics, and today different versions of set theory are used.

Example 1.33 (Monoids). The theory of monoids consists of the above-mentioned signature for monoids and the following axioms:

- associativity: $\forall x\, \forall y\, \forall z\; x \circ (y \circ z) \doteq (x \circ y) \circ z$,
- left-neutrality: $\forall x\; e \circ x \doteq x$,
- right-neutrality: $\forall x\; x \circ e \doteq x$.

Example 1.34 (Groups). The theory of groups consists of the above-mentioned signature for groups, all the axioms of the theory of monoids, and the following axioms:

- left-inverseness: $\forall x\; x \circ x^{-1} \doteq e$,
- right-inverseness: $\forall x\; x^{-1} \circ x \doteq e$.

Example 1.35 (Natural numbers). The theory of natural numbers is a tricky one, which we will come back to later (see Ex. 3.54 and 8.45). For now we can define it like this:

- a nullary function symbol 0,
- a unary function symbol *succ*,
- an axiom for the injectivity of *succ*: $\forall x\, \forall y\; (succ(x) \doteq succ(y) \rightarrow x \doteq y)$,
- an axiom that makes 0 the starting point: $\neg\exists x\; 0 \doteq succ(x)$.
- a set of axioms for induction: For every formula F with $FV(F) = \{x\}$, the axiom

$$\big(F(0) \wedge \forall\, x\big(F(x) \rightarrow F(succ(x))\big)\big) \rightarrow \forall x\; F(x).$$

These are Peano's axioms from 1889 [**?**]. As for set theory, they were written at a time when researchers only began to study the notions of logic and formal syntax. In particular, Peano did not know FOLEQ as we do today.

Example 1.36 (Natural numbers). A variant of the theory of natural numbers – usually called Peano arithmetic – extends Ex. 1.35 with:

- binary function symbols $+$ and $\cdot$,
- two axioms that define addition inductively,
- two axioms that define multiplication inductively.

Chapter 2

Context-Sensitive Constraints

2.1 Inference Systems

Definition 2.1 (Inference System/Calculus). An *inference system* consists of a set of *judgments* and a set of (inference) *rules*. A rule must be of the form

$$\frac{J}{J_1 \quad \dots \quad J_n} R$$

where $J, J_1, \dots, J_n$ are judgments. R is the (optional) name, J the *conclusion*, and $J_1, \dots, J_n$ the *hypotheses* of the rule. If $n = 0$, the rule is called an *axiom*.

The intuition of a rule as above is that we can derive/prove/conclude that J holds if we have already derived/proved/-concluded that $J_1, \dots, J_n$ hold.

Definition 2.2 (Derivation). A *derivation* in an inference system is a tree in which

- every node N is labelled with a judgment $J(N)$,
- for every node N with children $N_1, \dots, N_r$, the inference system contains the rule

$$\frac{J(N)}{J(N_1) \quad \dots \quad J(N_r)}$$

A judgment J *holds* iff there is a derivation whose root is labelled with J.

Remark 2.3. Note that the leaves of a derivation must be labelled with axioms.
The intuition of a derivation is that it is the proof/evidence/justification of a judgment. Judgements that occur as the root of a derivation are said to be established/derived/proved, to hold, or to be true.
The intuition of an inference system is that it defines which judgments hold.

Remark 2.4 (Relation to Parse Trees). A derivation over an inference system I is the same as a parse tree over the grammar containing one production

$$D ::= R(\underbrace{D, \dots, D}_{n})$$

for every rule R of I with n hypotheses.

2.2 Inference Systems for Well-Formed Syntax

Logical languages are usually context-sensitive. Therefore, logic has developed a specific method for defining context-sensitive languages, which uses a pair of a grammar and an inference system. The inference system uses a context.

As a first inference system, we give a supplementary definition to Def. 1.5.

Definition 2.5. For the syntax of FOL, we use the following two judgments:

- If $\Gamma = \{x_1, \ldots, x_n\}$ is a Σ-context and t is a term, the judgment

  $$\Gamma \vdash_\Sigma t : \texttt{term} \qquad \text{or} \qquad x_1, \ldots, x_n \vdash_\Sigma t : \texttt{term}$$

  (intended to mean that t is a well-formed Σ-term in context Γ).

- If $\Gamma = \{x_1, \ldots, x_n\}$ is a Σ-context and F is a formula, the judgment

  $$\Gamma \vdash_\Sigma F : \texttt{form} \qquad \text{or} \qquad x_1, \ldots, x_n \vdash_\Sigma F : \texttt{form}$$

  (intended to mean that F is a well-formed Σ-formula in context Γ).

The inference rules are given in Fig. 2.1.

$$\frac{\Gamma \vdash_\Sigma t_1 : \texttt{term} \quad \ldots \quad \Gamma \vdash_\Sigma t_n : \texttt{term} \quad f \in \Sigma_f \quad ar(f) = n}{\Gamma \vdash_\Sigma f(t_1, \ldots, f_n) : \texttt{term}} \qquad \frac{x \in \Gamma}{\Gamma \vdash_\Sigma x : \texttt{term}}$$

$$\frac{\Gamma \vdash_\Sigma t_1 : \texttt{term} \quad \ldots \quad \Gamma \vdash_\Sigma t_n : \texttt{term} \quad p \in \Sigma_p \quad ar(p) = n}{\Gamma \vdash_\Sigma p(t_1, \ldots, f_n) : \texttt{form}} \qquad \frac{\Gamma \vdash_\Sigma t_1 : \texttt{term} \quad \Gamma \vdash_\Sigma t_2 : \texttt{term}}{\Gamma \vdash_\Sigma t_1 \doteq t_2 : \texttt{form}}$$

$$\frac{\Gamma \vdash_\Sigma F : \texttt{form} \quad \Gamma \vdash_\Sigma G : \texttt{form}}{\Gamma \vdash_\Sigma F \wedge G : \texttt{form}} \qquad \frac{\Gamma \vdash_\Sigma F : \texttt{form} \quad \Gamma \vdash_\Sigma G : \texttt{form}}{\Gamma \vdash_\Sigma F \vee G : \texttt{form}} \qquad \frac{\Gamma \vdash_\Sigma F : \texttt{form} \quad \Gamma \vdash_\Sigma G : \texttt{form}}{\Gamma \vdash_\Sigma F \rightarrow G : \texttt{form}}$$

$$\frac{\Gamma \vdash_\Sigma F : \texttt{form}}{\Gamma \vdash_\Sigma \neg F : \texttt{form}} \qquad \frac{\Gamma, x \vdash_\Sigma F : \texttt{form}}{\Gamma \vdash_\Sigma \forall x\, F : \texttt{form}} \qquad \frac{\Gamma, x \vdash_\Sigma F : \texttt{form}}{\Gamma \vdash_\Sigma \exists x\, F : \texttt{form}}$$

Figure 2.1: Syntax of FOL

Lemma 2.6. *Assume a FOL-signature Σ and a Σ-context Γ.*

- $\Gamma \vdash_\Sigma t : \texttt{term}$ *holds* *iff* t *is a well-formed term over* Σ *with* $FV(t) \subseteq \Gamma$.
- $\Gamma \vdash_\Sigma F : \texttt{form}$ *holds* *iff* F *is a well-formed formula over* Σ *with* $FV(F) \subseteq \Gamma$.

Proof. Exercise. □

Chapter 3

Model-Theoretic Semantics

In the previous chapter, we saw the syntax of several logics. Even though the symbols used in the formulas had an intuitive meaning (We all knew that $\wedge$ is supposed to mean "and".), we avoided these intuitions. In this chapter, we will define formally the meaning of the symbols and the formulas. Instead of "meaning", it is also common to say *interpretation* or *semantics*.

The semantics of a context-free language is usually defined as a function from the syntax to some other language. This function is defined by induction on syntax trees. The "other language" is called the *meta-language*. For programming languages, the meta-language is often natural language. For example, the semantics of Java is given in various big books called something like "Java language specification". Another meta-language used to define the semantics of a programming language is assembler (or byte code in the case of Java). In that case the above-mentioned function is implemented in the compiler.

A crucial characteristic of logics is that the meta-language is a formal language as well, namely mathematics. There are many names for the function that maps syntax to semantics in the literature. We will use the function $[\![-]\!]$. Thus, $[\![w]\!]$ is the semantics of w.

Based on a context-free grammar for the syntax, $[\![-]\!]$ can be described as follows:

- Every non-terminal A of the syntax (e.g., `form` and `term`) is mapped to a set $[\![A]\!]$.
- Every word w (e.g., a formula or term, respectively) derived from A is mapped to an element $[\![w]\!] \in [\![A]\!]$.
- The latter is defined by induction on the syntax tree that derives w from A.

3.1 Semantics of Propositional Logic

3.1.1 The Standard Semantics

First we interpret the single non-terminal:

$$[\![\texttt{form}]\!] = \{0, 1\}$$

The intuition here is that for all formulas F, we have that $[\![F]\!] = 1$ denotes "F is true", and $[\![F]\!] = 0$ denotes "F is false". The elements 0 and 1 are called the *truth values*. Instead of $\{0, 1\}$, it is also common to use the sets $\{\bot, \top\}$ (bottom and top) or $\{F, T\}$.

Then we need a mapping $[\![-]\!] : \mathbf{Sen}^{PL}(\Sigma) \to \{0, 1\}$ for all signatures Σ. The definition is by induction on syntax trees, and we need one case for every production. The straightforward approach looks like this:

- $[\![\mathit{true}]\!] = 1$,
- $[\![\mathit{false}]\!] = 0$,
- $[\![F \wedge G]\!] = \min\{[\![F]\!], [\![G]\!]\} = \begin{cases} 1 & \text{if } [\![F]\!] = 1 \text{ and } [\![G]\!] = 1 \\ 0 & \text{otherwise} \end{cases}$,

- $[\![F \vee G]\!] = \max\{[\![F]\!], [\![G]\!]\} = \begin{cases} 1 & \text{if } [\![F]\!] = 1 \text{ or } [\![G]\!] = 1 \\ 0 & \text{otherwise} \end{cases}$,
- $[\![F \rightarrow G]\!] = \min\{[\![G]\!] - [\![F]\!], 0\} + 1 = \begin{cases} 1 & \text{if } [\![F]\!] \leq [\![G]\!] \\ 0 & \text{otherwise} \end{cases}$,
- $[\![\neg F]\!] = 1 - [\![F]\!] = \begin{cases} 1 & \text{if } [\![F]\!] = 0 \\ 0 & \text{otherwise} \end{cases}$,
- for $p \in \Sigma$: $[\![p]\!] = ???$

The above definition shows two things: Firstly, $\wedge$, $\vee$, $\rightarrow$, and $\neg$ can be interpreted naturally as "and", "or", "if ... then", and "not", respectively. Secondly, the boolean variables $p \in \Sigma$ cannot be interpreted naturally. This was to be expected: The boolean variables represent arbitrary statements, and of course arbitrary statements can be true or false depending on the situation.

Therefore, we cannot simply define a mapping $[\![-]\!] : \mathbf{Sen}(\Sigma) \rightarrow \{0, 1\}$. Instead, we introduce the concept of models.

Definition 3.1 (Models)**.** A *model* of the PL-signature Σ is a mapping $\Sigma \rightarrow \{0, 1\}$. We denote the set of PL-models for Σ by $\mathbf{Mod}^{PL}(\Sigma)$.

The idea of models is that they collect all the cases that are missing in the inductive definition of the semantics. Those are all cases for production that refer to the signature. In the case of PL, there is only one such production, namely `form` $::= p$. Therefore, PL-models consists of one function that interprets all $p \in \Sigma$. The semantics is then defined *relative to a model*, i.e., a formula has different meanings depending on the model in which it is interpreted. Instead of "model", it is common to say *interpretation function* or *structure*.

Now we can define the semantics of PL as follows.

Definition 3.2 (Semantics of PL)**.** Assume a PL-signature Σ and a model $I \in \mathbf{Mod}^{PL}(\Sigma)$. Then we define the interpretation $[\![-]\!]^I : \mathbf{Sen}(\Sigma) \rightarrow \{0, 1\}$ of Σ-sentences in I as follows:

- $[\![true]\!]^I = 1$,
- $[\![false]\!]^I = 0$,
- $[\![F \wedge G]\!]^I = \min\{[\![F]\!]^I, [\![G]\!]^I\}$,
- $[\![F \vee G]\!]^I = \max\{[\![F]\!]^I, [\![G]\!]^I\}$,
- $[\![F \rightarrow G]\!]^I = \min\{[\![G]\!]^I - [\![F]\!]^I, 0\} + 1$,
- $[\![\neg F]\!]^I = 1 - [\![F]\!]^I$,
- for $p \in \Sigma$: $[\![p]\!]^I = I(p)$.

Notation 3.3 (Satisfaction). We also write

$$I \models^{PL}_{\Sigma} F \quad \text{iff} \quad [\![F]\!]^I = 1$$

where PL and Σ are often omitted if they are clear from the context.
We say that I satisfies F or F holds in I.

Example 3.4. Assume the PL-signature $\Sigma = \{p, q, r\}$. A model $I \in \mathbf{Mod}^{PL}(\Sigma)$ is given by $I(p) = I(q) = 1$ and $I(r) = 0$. Then we have:

- $[\![p \wedge true]\!]^I = 1$,
- $[\![\neg(p \vee q) \vee r]\!]^I = 0$.

For some sentences, the semantics does not depend on the model. For example, we have $[\![true \wedge \neg false]\!]^I = 1$ and $[\![p \vee \neg p]\!]^I = 1$ in every model I. The study of such sentences is a primary concern of logic. Thus, they get a special name in the following definition.

Definition 3.5. Assume a PL-signature Σ and a Σ-sentence F. F is called

- a *theorem* or *tautology* if $[\![F]\!]^I = 1$ for all models I,
- *satisfiable* if $[\![F]\!]^I = 1$ for some (maybe all) models I,
- *unsatisfiable* or *contradiction* if $[\![F]\!]^I = 0$ for all models I.

Remark 3.6. One of the most important problems of computer science is the following: Given a PL-formula F, decide whether F is satisfiable or not. This problem is so important that it has its own name: SAT. It is a good exercise at this point to solve it, i.e., to give an algorithm that answers the above question for input F.

3.1.2 Other Semantics for the PL syntax

[1]

It is important to realize that a logic has a syntax and a semantics. This means that it is possible that two different logics have the same logic syntax (i.e., the same signatures and the same sentences) but different semantics. The following logics are not extremely important per se. But they serve as examples to illustrate that the semantics chosen above for the logic PL is not the only possible choice.

Example 3.7 (Fuzzy Propositional Logic). Fuzzy propositional logic (FPL) has the same syntax as PL. But $[\![\texttt{form}]\!] = [0;1]$, and a model for the signature Σ is a mapping $\Sigma \to [0;1]$. In other words, all real numbers between 0 and 1 are possible truth values. The intuition is that $[\![F]\!]^I = 0.4$ is that F is 40 % true. Thus, fuzzy logics permits us to talk about degrees of truth, insecure truths, and probabilities. Fuzzy logics have proved very useful in machine controls where interpretations often change gradually: For example, the rule "If another robot is *close*, decelerate *a little*." can could be used to avoid collisions of mobile robots.
The definition

$$[\![F \wedge G]\!]^I = \begin{cases} 1 & \text{if } [\![F]\!]^I = [\![G]\!]^I = 1 \\ 0 & \text{otherwise} \end{cases}$$

does not make sense anymore for fuzzy logics. But the definition

$$[\![F \wedge G]\!]^I = \min\{[\![F]\!]^I, [\![G]\!]^I\}$$

is still appropriate. Similarly, the other cases of Def. 3.2 can be reused.
Thus, if $[\![F]\!]^I = 0.4$ and $[\![G]\!]^I = 0.3$, then $[\![F \wedge G]\!]^I = 0.3$. There are also other fuzzy logics that use

$$[\![F \wedge G]\!]^I = [\![F]\!]^I \cdot [\![G]\!]^I.$$

Then we would have $[\![F \wedge G]\!]^I = 0.12$

Example 3.8 (Paraconsistent Propositional Logic). In paraconsistent logic, we use the same syntax as for PL, but put $[\![\texttt{form}]\!] = \{0, 1, ?, !\}$. The intuition is that $[\![F]\!]^I = ?$ means that the interpretation of F is unknown. The intuition of $[\![F]\!]^I = !$ means that F is both true and false. Paraconsistent logics have proved useful when talking about knowledge about the world: Often we do not know whether something is true or false, or we have arguments in favor of and against a certain conclusion.

3.2 Semantics of First-Order Logic

In principle, the semantics of FOLEQ and FOL is defined in the same way as the semantics of PL. The difference is that now we have two non-terminals that we must interpret (`form` and `term`) and that the signature has function

[1]This section can be skipped.

and predicate symbols that need a semantics from the models. Again, we start with the straightforward approach without using a model and see how far we get. The cases where we get stuck and need the models are written in gray.

- $[\![\texttt{form}]\!] = \{0,1\}$,
- $[\![\texttt{term}]\!] = \texttt{term}^I$ where $\texttt{term}^I$ is some set that must come from the model
- $[\![\mathit{true}]\!] = 1$,
- $[\![\mathit{false}]\!] = 0$,
- $[\![x]\!] =$ some element of $\texttt{term}^I$ that must come from the model or from somewhere else
- $[\![f(t_1,\ldots,t_n)]\!] = f^I([\![t_1]\!],\ldots,[\![t_n]\!])$ where f^I is some mapping that must come from the model
- $[\![p(t_1,\ldots,t_n)]\!] = p^I([\![t_1]\!],\ldots,[\![t_n]\!])$ where p^I is some mapping that must come from the model
- $[\![F \wedge G]\!] = \min\{[\![F]\!],[\![G]\!]\}$,
- $[\![F \vee G]\!] = \max\{[\![F]\!],[\![G]\!]\}$,
- $[\![F \to G]\!] = \min\{[\![G]\!] - [\![F]\!], 0\} + 1$,
- $[\![\neg F]\!] = 1 - [\![F]\!]$,
- $[\![s \doteq t]\!] = \begin{cases} 1 & \text{if } [\![s]\!] = [\![t]\!] \\ 0 & \text{otherwise} \end{cases}$
- $[\![\forall x\ F]\!] = \min\{[\![F]\!](u) \mid u \in \texttt{term}^I\} = \begin{cases} 1 & \text{if } [\![F]\!](u) = 1 \text{ for all } u \in \texttt{term}^I \\ 0 & \text{otherwise} \end{cases}$
  where $[\![F]\!](u)$ somehow means to evaluate $[\![F]\!]$ for the input u,
- $[\![\exists x\ F]\!] = \max\{[\![F]\!](u) \mid u \in \texttt{term}^I\} = \begin{cases} 1 & \text{if } [\![F]\!](u) = 1 \text{ for some } u \in \texttt{term}^I \\ 0 & \text{otherwise} \end{cases}$
  where $[\![F]\!](u)$ somehow means to evaluate $[\![F]\!]$ for the input u.

Inspecting the above definition attempt, we find that a model I must give us a set $\texttt{term}^I$, and functions f^I and p^I for all function symbols f and predicate symbols p. We also need a way to interpret variables. The latter will be taken care of by assignments. Thus, we define as follows.

Definition 3.9 (Models)**.** A *model* I of the $\mathcal{FOLEQ}$-signature $\Sigma = (\Sigma_p, \Sigma_f, ar)$ is a mapping with domain $\{\texttt{term}\} \cup \Sigma_f \cup \Sigma_p$ such that

- $\texttt{term}^I$ is a set (called the *universe* or the *carrier set*),
- for every $f \in \Sigma_f$ with $ar(f) = n$: f^I is a mapping $f^I : (\texttt{term}^I)^n \to \texttt{term}^I$,
- for every $p \in \Sigma_p$ with $ar(p) = n$: p^I is a mapping $p^I : (\texttt{term}^I)^n \to \{0,1\}$.

Here and in the sequel, we write $\texttt{term}^I$, f^I, p^I instead of $I(\texttt{term})$, $I(f)$, $I(p)$.
We denote the collection of Σ-models by $\mathbf{Mod}^{\mathcal{FOLEQ}}(\Sigma)$.

The FOL-models are the same as the $\mathcal{FOLEQ}$-models. This makes sense because $\mathcal{FOL}$ and $\mathcal{FOLEQ}$ also have the same signatures.

Definition 3.10 (Assignments)**.** Let Σ be a signature.

- An *assignment* α for a set of variables Γ and the Σ-model I is a mapping $\alpha : \Gamma \to \texttt{term}^I$.
- If $\Gamma = \{x_1,\ldots,x_n\}$ and $\alpha(x_i) = u_i$, we write α as $[x_1/u_1,\ldots,x_n/u_n]$.
- If α is an assignment for Γ, we write $[\alpha, x/u]$ for the assignment for $\Gamma \cup \{x\}$ that maps x to u and all other variables $y \in \Gamma$ to $\alpha(y)$.

Remark 3.11 (Substitutions and Assignments). Substitutions and assignments behave very similarly and have very similar notations. Assignments replace variables with semantic values of some model (i.e., elements of $\mathtt{term}^I$) in the same way in which substitutions replace variables with syntactic terms of some other context.

If we have a model and an assignment, we can define the semantics of FOLEQ-formulas and terms. The definition follows the inductive definition of the syntax.

Definition 3.12 (Semantics of FOLEQ)**.** Given

- a FOLEQ-signature Σ and
- a Σ-context Γ,

we define the interpretation $[\![w]\!]^{I,\alpha}$ of a formula or term w over Σ in context Γ

- in a Σ-model I and
- under an assignment α for Γ into I

as follows by induction on Γ and w.

- $[\![\mathtt{form}]\!]^{I,\alpha} = \{0,1\}$,
- $[\![\mathtt{term}]\!]^{I,\alpha} = \mathtt{term}^I$,
- $[\![\mathit{true}]\!]^{I,\alpha} = 1$,
- $[\![\mathit{false}]\!]^{I,\alpha} = 0$,
- $[\![x]\!]^{I,\alpha} = \alpha(x)$,
- $[\![f(t_1,\dots,t_n)]\!]^{I,\alpha} = f^I([\![t_1]\!]^{I,\alpha},\dots,[\![t_n]\!]^{I,\alpha})$
- $[\![p(t_1,\dots,t_n)]\!]^{I,\alpha} = p^I([\![t_1]\!]^{I,\alpha},\dots,[\![t_n]\!]^{I,\alpha})$
- $[\![F \wedge G]\!]^{I,\alpha} = \min\{[\![F]\!]^{I,\alpha}, [\![G]\!]^{I,\alpha}\}$,
- $[\![F \vee G]\!]^{I,\alpha} = \max\{[\![F]\!]^{I,\alpha}, [\![G]\!]^{I,\alpha}\}$,
- $[\![F \rightarrow G]\!]^{I,\alpha} = \min\{[\![G]\!]^{I,\alpha} - [\![F]\!]^{I,\alpha}, 0\} + 1$,
- $[\![\neg F]\!]^{I,\alpha} = 1 - [\![F]\!]^{I,\alpha}$,
- $[\![s \doteq t]\!]^{I,\alpha} = \begin{cases} 1 & \text{if } [\![s]\!]^{I,\alpha} = [\![t]\!]^{I,\alpha} \\ 0 & \text{otherwise} \end{cases}$,
- $[\![\forall x\, F]\!]^{I,\alpha} = \min\{[\![F]\!]^{I,[x/u,\alpha]} \mid u \in \mathtt{term}^I\}$,
- $[\![\exists x\, F]\!]^{I,\alpha} = \max\{[\![F]\!]^{I,[x/u,\alpha]} \mid u \in \mathtt{term}^I\}$.

In particular, for $\Gamma = \alpha = \varnothing$ and $F \in \mathbf{Sen}(\Sigma)$, we write $[\![F]\!]^I$ for the interpretation of F in I. Then we also write

$$I \models_\Sigma F \qquad \text{iff} \qquad [\![F]\!]^I = 1.$$

The semantics of FOL is the same as that of FOLEQ except that the case for $\doteq$ is dropped.

Example 3.13 (Monoids). Let Σ be the signature of monoids from above. A model $I \in \mathbf{Mod}^{FOLEQ}(\Sigma)$ consists of

- a set $\mathtt{term}^I$,
- an element $e^I \in \mathtt{term}^I$,

- a binary mapping $\circ^I : \mathtt{term}^I \times \mathtt{term}^I \to \mathtt{term}^I$.

Models of such small signatures are typically written as $(\mathtt{term}^I, e^I, \circ^I)$.
For example, we can say $I_1 = (\mathbb{Z}, 0, +)$ is a Σ-model. I_1 has the further properties that it satisfies all the axioms of the theory of monoids:

- associativity: $I_1 \models_\Sigma \forall x\, \forall y\, \forall z\; x \circ (y \circ z) \doteq (x \circ y) \circ z$,
- left-neutrality: $I_1 \models_\Sigma \forall x\; e \circ x \doteq x$,
- right-neutrality: $I_1 \models_\Sigma \forall x\; x \circ e \doteq x$.

There are lots of other Σ-models, for example, $I_2 = (\mathbb{N}, 0, -)$. While I_2 is a Σ-model, it does not satisfy all the axioms of the theory of monoids:

- associativity does not hold: $I_2 \not\models_\Sigma \forall x\, \forall y\, \forall z\; x \circ (y \circ z) \doteq (x \circ y) \circ z$,
- left-neutrality does not hold: $I_2 \not\models_\Sigma \forall x\; e \circ x \doteq x$,
- right-neutrality, however, holds: $I_2 \models_\Sigma \forall x\; x \circ e \doteq x$.

As an exercise, you should apply the definition of the semantics step by step to verify that the above statements about I_1 and I_2 are correct.

Example 3.14 (The Empty Model). For every signature, there is exactly one model with $\mathtt{term}^I =$. This is because the interpretation of all function and predicate symbols is uniquely determined to be the empty mapping. We call it the *empty* model.
The empty model E has some special properties: We always have $[\![\forall x.F]\!]^{E,\alpha} = 1$ and $[\![\exists x.F]\!]^{E,\alpha} = 0$. In particular, it is the only model that satisfies the formula $\forall x.\mathit{false}$.

Example 3.15 (The Trivial Model). For every signature, models T with $|\mathtt{term}^T| = 1$ are called *trivial*. Once the universe is fixed, e.g., $\mathtt{term}^T = \{0\}$, the interpretation of all function symbols f is uniquely determined: $f^I(0, \ldots, 0) = 0$. Similarly, there are only two options for every predicate symbol p: $p^I(0, \ldots, 0) \in \{0, 1\}$.
Thus, if there are no predicate symbols in Σ, there is essentially only 1 trivial model.

Remark 3.16 (The Empty Model). In Def. 3.9, we allow $\mathtt{term}^I =$. This is *not* common for $\mathcal{FOLEQ}$: In fact, virtually every textbook requires $\mathtt{term}^I \neq \varnothing$.
We allow the empty model because almost all results about $\mathcal{FOLEQ}$ still work for $\mathtt{term}^I =$ and many results become more elegant if it is allowed.
See also Rem. 3.27 and 4.4.

Summary We can summarize the syntax and semantics of *symbols* as follows:

Symbol	Syntax	Semantics
logical symbol (here: $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\forall$, $\exists$, $\doteq$)	always present	fixed interpretation
function or predicate symbol	declared (globally) in signature	interpreted by model
variable	declared (locally) in context	interpreted by assignment

Similarly, for a Σ-model I and an assignment α for a Σ-context Γ, we can summarize the syntax and semantics of *composed expressions* as follows:

Expression	Syntax	Semantics
term t	$\Gamma \vdash_\Sigma t : \mathtt{term}$	$[\![t]\!]^{I,\alpha} \in [\![\mathtt{term}]\!]^I$
formula F	$\Gamma \vdash_\Sigma F : \mathtt{form}$	$[\![F]\!]^{I,\alpha} \in [\![\mathtt{form}]\!]^I = \{0, 1\}$

This can be interpreted as saying that $[\![-]\!]$ (recursively) translates every judgment holding about the syntax to a judgment holding about the semantics: Whenever $\Gamma \vdash_\Sigma t : \mathtt{term}$, then $[\![t]\!]^{I,\alpha} \in [\![\mathtt{term}]\!]^I$, and accordingly for formulas.

3.3 Semantics of Contexts and Substitution

2

A substitution γ lets us move terms and formulas from a context Γ to a context Γ'. Often we are in a situation where we know the interpretation of F in context Γ and want to compute the interpretation of $\overline{\gamma}(F)$ in context Γ'. Of course, this is straightforward: apply the substitution γ to F, then use the definition of the semantics to interpret $\overline{\gamma}(F)$. But this requires two inductions on the syntax, and that can be quite complicated if F is a long formula.

Imagine we have a sentence $\forall x\ F(x)$, and for a certain model I we have already computed $[\![F(x)]\!]^{I,\alpha}$ for all assignments α in order to determine whether $I \models_\Sigma \forall x\ F(x)$. This can be very hard because if $\texttt{term}^I$ is infinite, there are infinitely many assignments. Now we want to determine whether $\forall y\ F(t(y))$. For that, we need to compute $[\![F(t(y))]\!]^{I,\alpha}$ for all assignments α. $F(t(y))$ arises from $F(x)$ by substituting x with $t(y)$. We want to use this fact and reuse the work we have already done.

This is possible by giving a semantics to substitutions and then studying how the interpretations of F, γ, and $\overline{\gamma}(F)$ are interrelated.

Definition 3.17 (Semantics of Contexts)**.** Assume a FOLEQ-signature Σ, a Σ-model I, and a Σ-context Γ. We define the interpretation of Γ in I as follows: $[\![\Gamma]\!]^I$ is the set of assignments for Γ into I.

Definition 3.18 (Semantics of Substitutions)**.** Assume a FOLEQ-signature Σ, a Σ-model I, and a Σ-substitution $\gamma : \Gamma \to \Gamma'$. We define the interpretation of γ in I as follows:

$$[\![\gamma]\!]^I : [\![\Gamma']\!]^I \to [\![\Gamma]\!]^I,$$

$$[\![\gamma]\!]^I : \alpha' \mapsto \alpha \text{ where } \alpha(x) = [\![\overline{\gamma}(x)]\!]^{I,\alpha'}.$$

This looks confusing. Let us "type-check" the definition, i.e., check whether $[\![\gamma]\!]^I$ is indeed a mapping from $[\![\Gamma']\!]^I$ to $[\![\Gamma]\!]^I$:

1. The input to $[\![\gamma]\!]^I$ is an assignment $\alpha' \in [\![\Gamma']\!]^I$. Thus, α' is an assignment for Γ' into I.
2. The output of $[\![\gamma]\!]^I$ is supposed to be an assignment $\alpha \in [\![\Gamma]\!]^I$. Thus, we have to prove that α is an assignment for Γ into I.
3. Thus, we have to prove that α is a mapping from Γ to $\texttt{term}^I$. So let us assume $x \in \Gamma$. We have to prove that $\alpha(x) \in \texttt{term}^I$.
   (a) $x \in \Gamma$ and $\gamma : \Gamma \to \Gamma'$. Therefore, $\overline{\gamma}(x)$ is a term in context Γ'.
   (b) We know α' is an assignment for Γ' into I.
   (c) Therefore, we can compute $[\![\overline{\gamma}(x)]\!]^{I,\alpha'}$. And the result is an element of $\texttt{term}^I$.
   (d) Thus, we have proved $\alpha(x) \in \texttt{term}^I$.
4. Thus, we have proved that $\alpha = [\![\gamma]\!]^I(\alpha')$ is an element of $[\![\Gamma]\!]^I$.

So the semantics of substitutions is indeed well-defined. It maps assignments for Γ' to assignments for Γ. Syntactically, a substitution $\gamma : \Gamma \to \Gamma'$ translates from Γ to Γ': By applying γ, we move Γ-terms and formulas to Γ'. Now we have seen that semantically γ is also a translation, but in the opposite direction: It maps assignments for Γ' to assignments for Γ. This is a very general effect that we find for all logics. Researchers have only fully understood it within the last decades. And understanding it has led to tremendous insights into the relation between syntax and semantics.

Using the interpretation of contexts and substitutions, we can obtain a different perspective on the interpretation of terms and formulas. Fix a signature Σ and a model I. Take formula F in context Γ. We cannot write $[\![F]\!]^I$

[2]This section can be skipped.

because F has free variables — we need an assignment for Γ, and then we have $[\![F]\!]^{I,\alpha} \in \{0,1\}$. In other words: If F is fixed, then every assignment α gives us an element of $\{0,1\}$. Thus, we can define

$$[\![F]\!]^I : [\![\Gamma]\!]^I \to \{0,1\},$$
$$[\![F]\!]^I : \alpha \mapsto [\![F]\!]^{I,\alpha}.$$

Similarly, we can define for a term t in context Γ:

$$[\![t]\!]^I : [\![\Gamma]\!]^I \to \texttt{term}^I,$$
$$[\![t]\!]^I : \alpha \mapsto [\![t]\!]^{I,\alpha}.$$

Thus, we can think of the interpretation of a formula in context Γ as a mapping from the set of assignments (i.e., the interpretation of Γ) to $\{0,1\}$. And we can think of the interpretation of a term in context Γ as a mapping from the set of assignments to $\texttt{term}^I$.

We can summarize this as follows:

Syntactically	Semantically
$\Gamma \vdash_\Sigma F : \texttt{form}$	$[\![F]\!]^I : [\![\Gamma]\!]^I \to [\![\texttt{form}]\!]^I = \{0,1\}$
$\Gamma \vdash_\Sigma t : \texttt{term}$	$[\![t]\!]^I : [\![\Gamma]\!]^I \to [\![\texttt{term}]\!]^I = \texttt{term}^I$
$\gamma : \Gamma \to \Gamma'$	$[\![\gamma]\!]^I : [\![\Gamma']\!]^I \to [\![\Gamma]\!]^I$

If we look at this table for a while, we get a tempting idea. What if we compose maps as follows:

$$\begin{tikzcd}
\texttt{term}^I & {[\![\Gamma]\!]^I} \arrow[l, "{[\![t]\!]^I}"'] \arrow[r, "{[\![F]\!]^I}"] & \{0,1\} \\
 & {[\![\Gamma']\!]^I} \arrow[ul, "{[\![\overline{\gamma}(t)]\!]^I}"] \arrow[u, "{[\![\gamma]\!]^I}"] \arrow[ur, "{[\![\overline{\gamma}(F)]\!]^I}"'] &
\end{tikzcd}$$

Could it be that the compositions yield exactly $[\![\overline{\gamma}(t)]\!]^I$ and $[\![\overline{\gamma}(F)]\!]^I$? Indeed they do.

Theorem 3.19 (Semantics of Substitution). *Assume a FOLEQ-signature Σ and a Σ-model I. Then for every substitution $\gamma : \Gamma \to \Gamma'$ and every formula F in context Γ and every term t in context Γ:*

$$[\![\overline{\gamma}(t)]\!]^I = [\![t]\!]^I \circ [\![\gamma]\!]^I,$$
$$[\![\overline{\gamma}(F)]\!]^I = [\![F]\!]^I \circ [\![\gamma]\!]^I.$$

Proof. By context-sensitive induction on the syntax tree of t and F. □

Thus, the semantics of substitution is simply composition.

3.4 An Abstract Definition of the Model Theory of a Logic

Above we have seen the semantics for PL, FOL, and FOLEQ. In Sect. 1.4, we found that the syntax of a logic can be abstractly characterized by pairs (**Sig**, **Sen**), which we called a logic syntax. We can make a similar observation about the semantics.

Definition 3.20 (Model Theory)**.** A model theory for the logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ is a pair $(\mathbf{Mod}, \models)$ such that

- for every $\Sigma \in \mathbf{Sig}$, we have that $\mathbf{Mod}(\Sigma)$ is a collection of objects, called the Σ-*models*,
- for every $\Sigma \in \mathbf{Sig}$, we have that $\models \subseteq \mathbf{Sen}(\Sigma) \times \mathbf{Mod}(\Sigma)$ is a binary relation, called the *satisfaction relation*.

If $I \models_\Sigma F$, we say "F holds in I" or "I satisfies F" or "I makes F true". We also write $I \not\models_\Sigma F$ for the opposite. A four-tuple $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ of a logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ and a model theory $(\mathbf{Mod}, \models)$ for it is called a *model-theoretical* logic.

Remark 3.21. Model-theoretical logics were introduced as *institutions* in [**?**].

Example 3.22. Then we immediately get some example model theoretical logics:

- Propositional logic $PL = (\mathbf{Sig}^{PL}, \mathbf{Sen}^{PL}, \mathbf{Mod}^{PL}, \models^{PL})$.
- First-order logic with equality $FOLEQ = (\mathbf{Sig}^{FOLEQ}, \mathbf{Sen}^{FOLEQ}, \mathbf{Mod}^{FOLEQ}, \models^{FOLEQ})$. Note that $\mathbf{Sen}^{FOLEQ}(\Sigma)$ only contains the closed Σ-formulas; therefore, $[\![F]\!]^I$ only depends on the model I, and no assignments are needed when talking about $\models$.
- The logics FOL, ALGEQ, and HORNEQ are obtained from FOLEQ in the obvious way.

3.5 Theorems and Consequence (Model-Theoretically)

One of the most important advantages of the abstract definitions given by institutions is that the main definitions regarding theories can be done for all institutions at once.

Definition 3.23 (Models)**.** Given syntax and model theory $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ and a theory $(\Sigma; \Theta)$. We define: A Σ-model I is a $(\Sigma; \Theta)$-model iff

$$I \models_\Sigma F \qquad \text{for all } F \in \Theta.$$

We write $\mathbf{Mod}(\Sigma, \Theta)$ for the collection of models of (Σ, Θ). In other words:

$$\mathbf{Mod}(\Sigma; \Theta) = \{I \in \mathbf{Mod}(\Sigma) \mid I \models_\Sigma F \ \text{ for all } F \in \Theta\}.$$

Then we are ready to make some far-reaching definitions:

Definition 3.24 (Monoids etc.)**.** We define:

- A model of the FOLEQ-theory from Ex. 1.33 is called a *monoid*.
- A model of the FOLEQ-theory from Ex. 1.34 is called a *group*.
- Commutative groups, rings, rings with 1, commutative rings, fields, etc. are all defined similarly using the respective signatures and axioms.

Example 3.25 (Fields). The theory of fields is given by the following signature

- binary function symbols $+$ and $\cdot$,
- unary function symbols $-$ and inv (where we write $inv(x)$ as x^{-1}),
- nullary functions symbols 0 and 1,

and the following axioms

- associativity of addition: $\forall x\, \forall y\, \forall z\; x + (y + z) \doteq (x + y) + z$,
- left-neutrality of zero: $\forall x\; 0 + x \doteq x$,
- right-neutrality of zero: $\forall x\; x + 0 \doteq x$,
- left-inverseness of subtraction: $\forall x\; (-x) + x \doteq 0$,
- right-inverseness of subtraction: $\forall x\; x + (-x) \doteq 0$,
- commutativity of addition: $\forall x\, \forall y\; x + y \doteq y + x$,
- associativity of multiplication: $\forall x\, \forall y\, \forall z\; x \cdot (y \cdot z) \doteq (x \cdot y) \cdot z$,
- left-neutrality of one: $\forall x\; 1 \cdot x \doteq x$,
- right-neutrality of one: $\forall x\; x \cdot 1 \doteq x$,
- left-inverseness of division: $\forall x\; (\neg x \doteq 0 \to x^{-1} \cdot x \doteq 1)$,
- right-inverseness of division: $\forall x\; (\neg x \doteq 0 \to x \cdot x^{-1} \doteq 1)$,
- commutativity of multiplication: $\forall x\, \forall y\; x \cdot y \doteq y \cdot x$,
- left-distributivity of multiplication over addition: $\forall x\, \forall y\, \forall z\; x \cdot (y + z) \doteq (x \cdot y) + (x \cdot z)$,
- right-distributivity of multiplication over addition: $\forall x\, \forall y\, \forall z\; (y + z) \cdot x \doteq (y \cdot x) + (z \cdot x)$.

A *field* is a model of this theory. Important fields are $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ with the usual functions.
Note that:

- We can define rings, rings with 1 (sometimes called unital rings or just rings), and commutative rings by dropping symbols and axioms from the theory of fields.
- In mathematics, *inv* is actually only a partial function: It is undefined for 0. In FOLEQ, we cannot talk about undefinedness. Therefore, we need a total function, and then the axioms need to say $\forall x\; (\neg x \doteq 0 \to \ldots)$.
- Right-neutrality, right-inverseness, and right-distributivity are actually redundant because of commutativity.

The area of mathematics called *algebra* is devoted to the study of the model collections $\mathbf{Mod}(\Sigma;\Theta)$ where $(\Sigma;\Theta)$ are the theories from above, in particular the theories of groups, rings, and fields.

Definition 3.26 (Theorems)**.** Given a syntax and model theory $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ and a theory $(\Sigma;\Theta)$. We define:

- A Σ-sentence is a (model-theoretical) *theorem* of $(\Sigma;\Theta)$ if it is true in all (Σ,Θ)-models. In other words,

$$Thm(\Sigma;\Theta) = \{F \in \mathbf{Sen}(\Sigma) \mid I \models_\Sigma F \text{ for all } I \in \mathbf{Mod}(\Sigma,\Theta)\}.$$

- If $F \in Thm(\Sigma;\Theta)$, we also write

$$\Theta \models_\Sigma F.$$

  Then we also say that F is a (model-theoretical) *consequence* of Θ.

Remark 3.27 (Non-Theorems due to the Empty Universe). Because we allow the empty universe (see Rem. 3.16), we have more models than textbook definitions and therefore less theorems. In particular, $\exists x.true$ and $\forall x.F \Rightarrow \exists x.F$ hold in all models except for the empty model. Therefore, they are usually $\mathcal{FOLEQ}$-theorems but are not theorems with our definition.

Notation 3.28. Note that this means that the symbol $\models$ has two different but related meanings:

- $I \models_\Sigma F$ means that F is true in the model I. This is called *satisfaction*, and we say that I satisfies F.
- $\Theta \models_\Sigma F$ means that F is true in all models of (Σ, Θ). This is called *entailment*, and we say that Θ entails F.
- The relation between the two is:

$$\Theta \models_\Sigma F \qquad \text{iff} \qquad I \models_\Sigma F \quad \text{for all } I \quad \text{for which } I \models_\Sigma T \text{ for all } T \in \Theta.$$

  Thus, intuitively, $\Theta \models_\Sigma F$ means "F holds in all models in which all sentences in Θ hold." Or "If a model satisfies all sentences in Θ, then it also satisfies F." Or even more intuitively: "If all sentences in Θ hold, then F holds."

Definition 3.29. A theory $(\Sigma; \Theta)$ is called *closed* if $\Theta = \mathit{Thm}(\Sigma; \Theta)$. $(\Sigma; \mathit{Thm}(\Sigma; \Theta))$ is a closed theory that arises by adding all theorems to Θ. This theory is called the *closure* of $(\Sigma; \Theta)$.

Theorem 3.30. *We have the following basic properties of theories:*

1. *If* $\Theta' \subseteq \Theta$, *then* $\mathbf{Mod}(\Sigma; \Theta') \supseteq \mathbf{Mod}(\Sigma, \Theta)$.
2. *If* $\Theta' \subseteq \Theta$, *then* $\mathit{Thm}(\Sigma; \Theta') \subseteq \mathit{Thm}(\Sigma, \Theta)$.
3. $F \in \mathit{Thm}(\Sigma; \Theta)$ *for every* Σ*-tautology* F.
4. $\Theta \subseteq \mathit{Thm}(\Sigma; \Theta)$.
5. $\mathbf{Mod}(\Sigma; \mathit{Thm}(\Sigma; \Theta)) = \mathbf{Mod}(\Sigma; \Theta)$.
6. $\mathit{Thm}(\Sigma; \mathit{Thm}(\Sigma; \Theta)) = \mathit{Thm}(\Sigma; \Theta)$.

Proof.

1. Exercise.
2. Exercise.
3. Special case of the second result: Put $\Theta' = \varnothing$.
4. Assume $F \in \Theta$ (*). We have to show $F \in \mathit{Thm}(\Sigma; \Theta)$.
   (a) By definition of *Thm*, we have to show $I \models_\Sigma F$ for every model I such that $I \in \mathbf{Mod}(\Sigma; \Theta)$.
   (b) Assume a model $I \in \mathbf{Mod}(\Sigma; \Theta)$ (**). We have to show $I \models_\Sigma F$.
      i. By (**) and the definition of **Mod**, we have that $I \models_\Sigma G$ for every $G \in \Theta$.
      ii. By (*), we can put $G = F$ and obtain $I \models_\Sigma F$.
      iii. Done.
   (c) Done.
5. The $\subseteq$ direction follows by combining the first and the fourth result.
   To prove the $\supseteq$ direction, we verbalize the involved sets:
   - $\mathbf{Mod}(\Sigma; \Theta)$: the models satisfying all sentences in Θ
   - $\mathit{Thm}(\Sigma; \Theta)$: the sentences satisfied by all models satisfying all sentences in Θ
   - $\mathbf{Mod}(\Sigma; \mathit{Thm}(\Sigma; \Theta))$: the models satisfying all sentences satisfied by all models satisfying all sentences in Θ

   Then it is easy to see that $\mathbf{Mod}(\Sigma; \mathit{Thm}(\Sigma; \Theta)) \supseteq \mathbf{Mod}(\Sigma; \Theta)$

6. This follows immediately using the definition of *Thm* and the fifth result.

□

Now we introduce some important notions for theories.

Definition 3.31. For a fixed theory $(\Sigma;\Theta)$, a sentence $F \in \mathbf{Sen}(\Sigma)$ is called

- a *theorem*, a *tautology*, or *valid* if it holds in all $(\Sigma;\Theta)$-models,
- a *contradiction* or *unsatisfiable* if it holds in no $(\Sigma;\Theta)$-model.

The words "tautology" and "valid" are especially common for the special case $\Theta = \varnothing$.

Lemma 3.32. *If a logic has negation, then F is a theorem iff $\neg F$ is a contradiction and F is a contradiction iff $\neg F$ is a theorem.*

Proof. Clear because F holds in a model iff $\neg F$ does not hold in it, and vice versa. □

Example 3.33. Examples for tautologies are *true*, $\neg$*false*, $(F \wedge G) \to (G \wedge F)$, or $\neg\neg F \to F$. There is a number of important and fairly obvious tautologies. A good exercise is to find some more and check that they really are tautologies. An important less obvious tautology is this: *false* $\to F$ for any F.
The most important examples for contradictions are *false*, $F \wedge \neg F$ for any F, and $\neg F$ for any tautology F.

Thus, a theory splits the sentences into three groups: theorems (true in all models), contradictions (true in no model), and the rest (true in some but not all models). Thus, only the sentences in the third group differ between models – they distinguish the models. Depending on the size of the three groups, we distinguish two special cases of theories:

Definition 3.34. A theory $(\Sigma;\Theta)$ is called

- *inconsistent* if all sentences are theorems, $\Theta \models_\Sigma F$ for every $F \in \mathbf{Sen}(\Sigma)$,
- *consistent* if it is not inconsistent, i.e., there is a sentence that is not a theorem,
- *complete* if every sentence is either a theorem or a contradiction, i.e., if for every $F \in \mathbf{Sen}(\Sigma)$ either $\Theta \models_\Sigma F$ or $\Theta \models_\Sigma \neg F$.

Inconsistent theories make all sentences theorems including, e.g., *false*. Thus, truth becomes meaningless. For example, a theory is inconsistent if it contains *false* or any other contradiction as an axiom or if it contains both F and $\neg F$ as axioms. An inconsistent theory cannot have a model. Precisely, we have:

Lemma 3.35. *The following are equivalent for a theory $(\Sigma;\Theta)$ in any logic:*

1. *It has no model.*
2. *It is inconsistent.*
3. *(If negation is interpreted as for PL:) F and $\neg F$ are theorems for some F.*
4. *(If falsity is interpreted as for PL:) false is a theorem.*

Proof. By circular implications:
1 implies 2: If there is no model, then every sentence is a theorem.
2 implies 3: If every sentence is a theorem, then at least F and $\neg F$ for some F.
3 implies 4: No model can satisfy F and $\neg F$. So every model that does also satisfies *false*.
4 implies 1: No model satisfies *false*. □

Complete theories make "half the sentences" theorems: For every F either F or $\neg F$ is a theorem, and the other one is a contradiction. Since the third group is empty, models have no freedom how to interpret the sentences. If we add one more sentence to a complete theory, it becomes inconsistent. Precisely, we have:

Lemma 3.36. *The following are equivalent for a theory* $(\Sigma; \Theta)$*:*

1. *It is complete.*
2. *There are models, and all models satisfy exactly the same formulas.*
3. *There is a model satisfying exactly the theorems of* $(\Sigma; \Theta)$.

Proof. By circular implications:
1 implies 2: If there were no models, then the theory would be inconsistent, but completeness implies consistency. These models must satisfy at least all the theorems of $(\Sigma; \Theta)$. Due to completeness, if some model satisfied one further sentence, that sentence would have to be a contradiction, which is impossible.
2 implies 3: Since all models satisfy the same sentences, those are the theorems of $(\Sigma; \Theta)$. Since there are models, we can pick any one of them, and that will satisfy exactly the theorems of $(\Sigma; \Theta)$.
3 implies 1: For any model I, the set $\{F \mid I \models_\Sigma F\}$ of sentences it satisfies is a complete theory. So $(\Sigma; \Theta)$ must be complete. □

Notation 3.37. We have defined the notions "consequence", "theorem", "contradiction", "(in)consistent", and "complete" model-theoretically. All have proof-theoretical analogues, see Def. 4.10 and 4.12. If we need to distinguish them, we will prefix M, e.g., we will say that a theory is M-consistent. See also Not. 4.14.

3.6 Specification and Abstract Data Types

The process of *specification* or *axiomatization* is the following: Given a class $\mathcal{M} \subseteq \mathbf{Mod}(\Sigma)$ of models, find a theory $\Theta \subseteq \mathbf{Sen}(\Sigma)$ such that $\mathcal{M}$ contains exactly the models of Θ.
Then we call the triple $(\Sigma, \Theta, \mathcal{M})$ an abstract data type.

3.6.1 Theoretical Foundation

[3]

Definition 3.38 (Model Class)**.** Assume a syntax-model theory pair $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$. A model class is a pair $(\Sigma; \mathcal{M})$ where $\Sigma \in \mathbf{Sig}$ and $\mathcal{M} \subseteq \mathbf{Mod}(\Sigma)$.

Model classes are the analogue to theories. Just like for theories, we can define two important operations.

Definition 3.39. For every model class $(\Sigma, \mathcal{M})$, we define the theory of $(\Sigma; \mathcal{M})$ by

$$\mathit{Thy}(\Sigma; \mathcal{M}) = \{F \in \mathbf{Sen}(\Sigma) \mid I \models_\Sigma F \text{ for all } I \in \mathcal{M}\}$$

and the closure of M by

$$\mathit{ModC}(\Sigma; \mathcal{M}) = \{I \in \mathbf{Mod}(\Sigma) \mid I \models_\Sigma F \text{ for all } F \in \mathit{Thy}(\Sigma; \mathcal{M})\}$$

Model classes satisfying $\mathit{ModC}(\Sigma; \mathcal{M}) = \mathcal{M}$ are called closed.

Notation 3.40. Let us now fix a signature Σ and use the following abbreviations:

- Θ^* for $\mathbf{Mod}(\Sigma; \Theta)$,
- $\mathcal{M}^*$ for $\mathit{Thy}(\Sigma; \mathcal{M})$,

[3]This section can be skipped.

Then we have the following mappings:

$$\mathcal{P}(\mathbf{Sen}(\Sigma)) \to \mathcal{P}(\mathbf{Mod}(\Sigma)), \qquad \Theta \mapsto \Theta^*$$

$$\mathcal{P}(\mathbf{Mod}(\Sigma)) \to \mathcal{P}(\mathbf{Sen}(\Sigma)), \qquad \mathcal{M} \mapsto \mathcal{M}^*$$

$$\mathcal{P}(\mathbf{Sen}(\Sigma)) \to \mathcal{P}(\mathbf{Sen}(\Sigma)), \qquad \Theta \mapsto \Theta^{**}$$

$$\mathcal{P}(\mathbf{Mod}(\Sigma)) \to \mathcal{P}(\mathbf{Mod}(\Sigma)), \qquad \mathcal{M} \mapsto \mathcal{M}^{**}$$

and we have

$$\Theta^{**} = \mathit{Thm}(\Sigma;\Theta)$$

$$\mathcal{M}^{**} = \mathit{ModC}(\Sigma;\mathcal{M})$$

Intuitively, $-^*$ turns a set of sentences into a collection of models (namely those satisfying all the sentences). It also maps the other way round: Every class of models is mapped to a set of sentences (namely those satisfied by all the models). Thus, $-^*$ maps back and forth between theories and model classes.

We can compose these mappings. If we start with a theory and go to model classes and back, i.e., we apply $-^*$ twice to $(\Sigma;\Theta)$, we obtain a new theory. And we already know this theory: It is $(\Sigma;\mathit{Thm}(\Sigma;\Theta))$. If we compose the other way round, we start with a model class $\mathcal{M}$, go to its theory, then take that theory's models. This composition also maps a model class $(\Sigma;\mathcal{M})$ to a larger one: It contains all the models that satisfy the same sentences as those in $\mathcal{M}$.

This is an instance of a very general concept called a Galois connection. Every binary relation $\rho \subseteq A \times B$ induces a Galois connection between $\mathcal{P}(A)$ and $\mathcal{P}(B)$. In our case $A = \mathbf{Sen}(\Sigma)$, $B = \mathbf{Mod}(\Sigma)$, and $\rho\ = \models_\Sigma$.

We have the following properties:

- $\Theta \subseteq \Theta'$ implies $\Theta^* \supseteq \Theta'^*$, and $\mathcal{M} \subseteq \mathcal{M}'$ implies $\mathcal{M}^* \supseteq \mathcal{M}'^*$,
- $\Theta \subseteq \Theta^{**}$ and $\mathcal{M} \subseteq \mathcal{M}^{**}$,
- $\Theta \subseteq \Theta'$ implies $\Theta^{**} \subseteq \Theta'^{**}$, and $\mathcal{M} \subseteq \mathcal{M}'$ implies $\mathcal{M}^{**} \subseteq \mathcal{M}'^{**}$,
- $\Theta^{**} = (\Theta^{**})^{**}$ and $\mathcal{M}^* = (\mathcal{M}^{**})^{**}$.
- Θ^* and $\mathcal{M}^*$ are closed.
- The closed theories are in bijection to the closed model classes.

Thus, specification means to find a Θ such that $\Theta^* = \mathcal{M}$. Often we are interested in the special case $\mathcal{M} = \{I\}$ for some intended model I.

It cannot always be possible to axiomatize a class of models: Θ^* is always a closed model class. So let us assume that $\mathcal{M}$ is closed. If $\mathcal{M}$ is closed, we can always find an axiomatization Θ by simply putting $\Theta := \mathcal{M}^*$.

But that does not help much: Θ should be as simple as possible. Ideally, Θ should be finite. But if Θ is infinite, it should at least be decidable (i.e., we should have an algorithm that tells us which axioms are in Θ). Thus, more precisely, specification means to find a simple Θ such that $\Theta^* = \mathcal{M}$.

Therefore, when we do specification, two things can go wrong:

- $\mathcal{M}$ is not closed, i.e., there is no appropriate Θ.
- There is a Θ but no simple one.

3.6.2 Examples of Abstract Data Types

Some of the most important and prevalent data types of mathematics are specified as finite FOLEQ theories. We have already seen some in Ex. 3.25. This section gives many more examples.

This approach of applying logic to systematically define data types in mathematics, specifically algebra, in terms of logic and model theory goes back to work by Robinson [**?**].

No Symbols

The empty signature yields the data type of sets.

Without any function or predicate symbols, all elements of the universe are interchangeable. Indeed, the only atomic formulas are equalities between variables. Therefore, the only property $\mathcal{FOLEQ}$ can talk about is the cardinality of the universe. The following axioms can be used to specify cardinality:

- universe has at least n elements:

$$L_n \quad = \quad \exists x_1 \ldots, \exists x_n. \bigwedge_{i,j=1,\ldots,n,\ i\neq j} \neg x_i \doteq x_j$$

  where we use $\bigwedge$ to abbreviate a long conjunction
- universe has at most n elements:

$$M_n \quad = \quad \exists x_1 \ldots, \exists x_n. \forall y. y \doteq x_1 \vee \ldots \vee y \doteq x_n$$

As special cases, we obtain

- universe is empty: $M_0 = \forall x.\mathit{false}$
- universe is non-empty: $L_1 = \exists x.\mathit{true}$

The data type of infinite sets can be specified using the infinite theory $\{L_1, L_2, \ldots\}$. The data type of finite sets cannot be specified in $\mathcal{FOLEQ}$.

One Binary Function Symbol

The simplest data types are those of binary function symbols. Addition, multiplication, etc. are typical examples.

Example 3.41 (Data types based on a binary function symbol). Fig. 3.1 gives the most important data types based for a binary function symbol.

symbol or axiom		magma	semigroup	commutative semigroup	monoid	commutative monoid	group	Abelian group	semilattice
$\circ$, binary function, written $s \circ t$	composition	✓	✓	✓	✓	✓	✓	✓	✓
e, nullary function	element				✓	✓	✓	✓	
inv, unary function, written t^{-1}	inverse element						✓	✓	
$\forall x \forall y \forall z\ (x \circ y) \circ z \doteq x \circ (y \circ z)$	associative		✓	✓	✓	✓	✓	✓	✓
$\forall x\ (x \circ e \doteq x \wedge e \circ x \doteq x)$	neutral element				✓	✓	✓	✓	
$\forall x\ (x \circ e \doteq e \wedge e \circ x \doteq e)$	absorbing element								
$\forall x\ (x \circ x^{-1} \doteq e \wedge x^{-1} \circ x \doteq e)$	inverse element						✓	✓	
$\forall x \forall y\ x \circ y \doteq y \circ x$	commutative			✓		✓		✓	✓
$\forall x\ x \circ x \doteq x$	idempotency								✓

Figure 3.1: Data types for a binary function symbols

Models of the data types from Ex. 3.41 occur all over mathematics and computer science:

Example 3.42 (Models based on a binary function symbol). We have the following models, where we write U for any number set, i.e., $U \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}\}$:

- Addition of numbers: $(U, +, 0)$ is a commutative monoid. Additionally, if $U \neq \mathbb{N}$, $(U, +, 0, -)$ is an Abelian group.
- Addition of other objects: It is a widely used convention to name operations $+$ and 0 and $-$ if they form commutative monoids or Abelian groups, e.g., addition of vectors, matrices, polynomials.
- Multiplication of numbers: $(U, \cdot, 1)$ is a commutative monoid. 0 is an annihilating element.
- Multiplication of other objects: It is a widely used convention to name operations $\cdot$ and 1 if they form monoids, e.g., multiplication of matrices, polynomials. However, these do not have to be commutative, e.g., for matrices.
- Maximum/minimum of numbers: If $U \neq \mathbb{C}$, $(U, \max)$ and $(U, \min)$ are semilattices. 0 is a neutral element for $(U, \max)$ and an annihilating element for $(U, \min)$.
- Modulus: In the magma $(\mathbb{N}, \mathrm{mod})$, the operation $a \mathrm{mod} b$ returns the smallest $m \in \mathbb{N}$ such that $b|(a-m)$. If $b \neq 0$, we can see m as the remainder left after dividing a by b; if $b = 0$, we have $m = a$. This magma has none of the properties mentioned above. However, with the definitions of Rem. 3.45, we have that 0 is a left-absorbing and a right-neutral element.
- Function composition: $(S^S, \circ, id_S)$ is a monoid; it is commutative iff $|S| \in \{0, 1\}$. If we only consider bijective functions, then inversion yields an inverse element and thus a group.
- Sets: $(\mathcal{P}(S), \cup, \varnothing, S)$ and $(\mathcal{P}(S), \cap, S, \varnothing)$ are commutative monoids with annihilating elements. They are also semilattices.
- Relations: As a special case of the power set example, we obtain the model of n-ary relations on $A_1, \ldots, A_n$ as $(\mathcal{P}(A_1 \times \ldots \times A_n), \cup, \varnothing, \cap, A_1 \times \ldots \times A_n)$.
- Binary endorelations: As a special case of the relations examples, we obtain the model of binary relations on A as $(\mathcal{P}(A \times A), \cup, \varnothing, \cap, A \times A)$. In this model, we can also define the diagonal $\Delta = \{(x, x) : x \in A\}$ and the composition $r; s = \{(x, z) \in A \times A \mid \text{ exists } y \text{ such that } (x, y) \in r,\ (y, z) \in s\}$. Then $(\mathcal{P}(A \times A), ;, \Delta, \varnothing)$ is also a monoid with annihilating element $\varnothing$.
- Concatenation of words over an alphabet: $(\Sigma^*, \cdot, \varepsilon)$ is a monoid where $\cdot$ is concatenation of words. It is commutative iff $|\Sigma| \in \{0, 1\}$.
- Operations on languages over an alphabet: $(\mathcal{P}(\Sigma^*), \cdot, \{\varepsilon\})$ is a monoid where $\cdot$ is concatenation of languages.

Remark 3.43 (Duality). By flipping the arguments of a binary function $\circ$, we obtain its dual $\circ^d$, i.e., $s \circ t$ is the same as $t \circ^d s$. If $\circ$ is commutative, than $\circ$ and $\circ^d$ are interchangeable.

We have the following correspondences:

$\circ$ is/has …	iff	$\circ^d$ is/has …
commutative		commutative
associative		associative
idempotent		idempotent
a neutral element e		a neutral element e
an absorbing element a		an absorbing element a
inverse elements x^{-1}		inverse elements x^{-1}

Remark 3.44 (Uniqueness). The neutral, absorbing, and inverse elements are uniquely determined if they exist. That is easy to prove: For example, if e and e' are neutral elements, then we can prove $e \doteq e \circ e'$ and $e \circ e' \doteq e'$.

Therefore, the pair of function symbol and axiom is often replaced with an existential axiom as follows:

function symbol and axiom	alternative axiom
nullary function symbol e, neutral element axiom for $\circ, e$	$\exists z \forall x\ (x \circ z \doteq x \wedge z \circ x \doteq x)$
nullary function symbol a, absorbing element axiom for $\circ, a$	$\exists z \forall x\ (x \circ z \doteq z \wedge z \circ x \doteq z)$
unary function symbol inv, inverse element axiom for $\circ, e, inv$	$\forall x \exists z (x \circ z \doteq e \wedge z \circ x \doteq e)$

This has the advantage that all data types have only a single function symbol and all differences are expressed as axioms.

Remark 3.45 (Left and Right Laws). The axioms for neutral, absorbing, and inverse elements involve a conjunction of two properties. Of course, if $\circ$ is commutative, both properties are equivalent and one of them can be omitted.

In the general case, we sometimes distinguish further between left- and right-neutral elements: These are elements that satisfy only one of the two properties (e.g., a left-neutral element satisfies $\forall x\ e \circ x \doteq x$). Left- and right-absorbing and left- and right-inverse elements are defined accordingly.

Now the uniqueness properties of Rem. 3.44 can be strengthened as follows: If a binary function has a left-neutral l element and a right-neutral element r, then they are equal ($l \doteq r$ is a theorem) and thus a neutral element. Accordingly, a left- and a right-absorbing element must be equal. However, it is possible (although rarely useful) to have multiple different left-neutral elements if there are no right-neutral elements. The according statement holds for the other cases.

Note that if an element e is both left-neutral and left-absorbing, then the model is trivial. The same holds if an element is both right-neutral and right-absorbing. But there are non-trivial models where an element is both left-neutral and right-absorbing (or right-neutral and left-absorbing).

For inverse elements, the corresponding statement requires that $\circ$ is associative: In that case, if there are left- and right-inverse elements, then they are equal and are inverse elements.

Two Binary Function Symbols

The next big class combines two function symbols. These often represent the interplay of, e.g., addition and multiplication.

Example 3.46 (Data types based on two binary function symbols). Fig. 3.2 gives datatypes that combine two of the datatypes from Ex. 3.46.

Additionally, we define

- the theory *ring without 1* by removing the 1 symbol and its axioms from the theory *ring*,
- the theory *lattice* by removing the 0 and 1 symbols and their axioms from the theory *bounded lattice*

Example 3.47 (Models based on two binary function symbols). Models of the data types from Ex. 3.46 are often obtained by combining models from Ex. 3.42.

We write the models as $(\mathtt{term}^I, \circ^I, 0^I, -^I, *^I, 1^I, inv^I)$ where we omit the components that are not applicable:

- Addition and multiplication: $(\mathbb{N}, +, 0, \cdot, 1)$ is a commutative semi-ring. $(\mathbb{Z}, +, 0, -, \cdot, 1)$ is a commutative ring. $(U, +, 0, -, \cdot, 1, ^{-1})$ are fields for $U \in \{\mathbb{Q}, \mathbb{R}, \mathbb{C}\}$.
- Matrices: $(R^{n\times n}, +, 0, -, \cdot, 1)$ is a ring if R is. (Note that we have to use square matrices so that multiplication is a total function.) It is usually not commutative even if R is.
- Multiplication: In the statement of distributivity, we say that $*$ distributes over $\circ$. For the natural numbers, multiplication distributes over various operations: $(\mathbb{N}, +, \cdot)$, $(\mathbb{N}, \mathrm{mod}, \cdot)$, $(\mathbb{N}, \min, \cdot)$, $(\mathbb{N}, \max, \cdot)$ are all distributive.
- Polynomials: $(R[X], +, 0, -, \cdot, 1)$ is a ring if R is; it is commutative if R is.
- Sets: For subsets x of S, the complement x^C is defined by $S \setminus x$. Then $(\mathcal{P}(S), \cup, \varnothing, \cap, S, -^C)$ is a Boolean lattice. Note that the special case $|S| = 1$ yields the usual 2-element Booleans.

[4] In addition to the axioms listed in this table, the theory of fields has the axiom $\neg 0 \doteq 1$.

[5] If present, the theory is called *commutative (semi-)ring*.

symbol and/or axiom		semi-ring	ring	field[4]	bounded lattice	Boolean lattice
binary functions $\circ$, $*$		✓	✓	✓	✓	✓
associativity for $\circ$		✓	✓	✓	✓	✓
associativity for $*$		✓	✓	✓	✓	✓
nullary function 0 and neutral element axiom for $\circ$, 0		✓	✓	✓	✓	✓
nullary function 1 and neutral element axiom for $*$, 1		✓	✓	✓	✓	✓
commutativity for $\circ$		✓	✓	✓	✓	✓
commutativity for $*$		optional[5]	optional[5]	✓	✓	✓
$\forall x \forall y \forall z\ x * (y \circ z) \doteq (x * y) \circ (x * z)$ $\forall x \forall y \forall z\ (y \circ z) * x \doteq (y * x) \circ (z * x)$	distributivity	✓	✓	✓		✓
unary function $-$ and inverse element axiom for $\circ$, 0, $-$			✓	✓		
$\forall x\ x * 0 \doteq 0 \wedge 0 * x \doteq 0$	absorbing element for $*, 0$	✓	T	T	✓	✓
$\forall x\ x \circ 1 \doteq 1 \wedge 1 \circ x \doteq 1$	absorbing element for $\circ, 1$				✓	✓
idempotency for $\circ$					✓	✓
idempotency for $*$					✓	✓
$\forall x \forall y\ x * (x \circ y) \doteq x$ $\forall x \forall y\ x \circ (x * y) \doteq x$	lattice absorption				✓	✓
unary function *inv*, written t^{-1}	inverse/complement			✓		✓
$\forall x\ \neg x \doteq 0 \Rightarrow (x * x^{-1} \doteq 1 \wedge x^{-1} * x \doteq 1)$	multiplicative inverse			✓		
$\forall x\ x \circ x^{-1} \doteq 1 \wedge x^{-1} \circ x \doteq 1$ $\forall x\ x * x^{-1} \doteq 0 \wedge x^{-1} * x \doteq 0$	complement (not inverse elements!)					✓

Figure 3.2: Data types for two binary function symbols (T marks theorems)

- Languages: $(\mathcal{P}(\Sigma^*), \cup, \varnothing, \cdot, \{\varepsilon\})$ is a semiring.
- The remainder classes of the integers: For $k \in \mathbb{N}$, let $\mathbb{Z}_k = \{0, \ldots, k-1\}$ if $k \neq 0$ and $\mathbb{Z}_k = \mathbb{Z}$ if $k = 0$. Let $+_k : \mathbb{Z}_k^2 \to \mathbb{Z}_k$, $-_k : \mathbb{Z}_k \to \mathbb{Z}_k$, and $*_k : \mathbb{Z}_k^2 \to \mathbb{Z}_k$ be like $+$, $-$, and $*$ but modulo k, e.g., $x +_k y = (x+y) \bmod k$. Then $(\mathbb{Z}_k, +_k, 0, -_k, *_k, 1)$ is a commutative ring. If k is prime, we can also define a multiplicative inverse and obtain a field.
- Binary endorelations: The binary relations on A form a semi-ring $(\mathcal{P}(A \times A), \cup, \varnothing, ;, \Delta)$. It is usually not commutative (unless $|A| \leq 1$).

One Binary Predicate Symbol

Similarly important data types use a binary predicate symbol and various combinations of axioms:

Example 3.48 (Data types based on a binary predicate symbol). Fig. 3.3 defines data types based on a binary predicate.

Of course, we usually use a different symbol than $\leq$ when a relation is symmetric, e.g., $\equiv$.

Additionally, we define

- the theory of orders bounded below/above by using only the axiom for the least/greatest element instead of

both,
- the theory of lattices by combining the axioms of meet- and join-semilattices,
- the theory of bounded lattices by combining the axioms of bounded orders and lattices.

symbol or axiom		relation	equivalence relation	preorder	order, partial order, poset	total order	dense order	bounded order	join-semilattice	meet-semilattice
$\leq$, written $s \leq t$	comparison	✓	✓	✓	✓	✓	✓	✓	✓	✓
$\forall x\ x \leq x$	reflexive		✓	✓	✓	✓	✓	✓	✓	✓
$\forall x \forall y \forall z\ ((x \leq y \wedge y \leq z) \Rightarrow x \leq z)$	transitive		✓	✓	✓	✓	✓	✓	✓	✓
$\forall x \forall y\ (x \leq y \Rightarrow y \leq x)$	symmetric		✓							
$\forall x \forall y\ ((x \leq y \wedge y \leq x) \Rightarrow x \doteq y)$	antisymmetric				✓	✓	✓	✓	✓	✓
$\forall x \forall y\ (x \leq y \vee y \leq x)$	total, linear					✓				
$\forall x \forall z\ (x < z \Rightarrow \exists y\ (x < y \wedge y < z))$ where $x < y$ abbreviates $x \leq y \wedge \neg x \doteq y$	dense						✓			
$\exists x \forall y\ x \leq y$	smallest element							✓		
$\exists x \forall y\ y \leq x$	greatest element							✓		
$\forall x \forall y \exists u\ (above(u,x,y) \wedge \forall u'\ (above(u',x,y) \Rightarrow u \leq u'))$ where $above(z,x,y)$ abbreviates $x \leq z \wedge y \leq z$	least upper bound, supremum					T			✓	
$\forall x \forall y \exists l\ (below(l,x,y) \wedge \forall l'\ (below(l',x,y) \Rightarrow l' \leq l))$ where $below(z,x,y)$ abbreviates $z \leq x \wedge z \leq y$	greatest lower bound, infimum					T				✓

Figure 3.3: Data types for a binary predicate symbols (T marks theorems)

Example 3.49 (Models based on a binary predicate). Models of the data types from Ex. 3.48 are also very common. We use U as in Ex. 3.42.

- Size of numbers: $(U, \leq)$ is a total order for $U \neq \mathbb{C}$. It is dense for $U \in \{\mathbb{Q}, \mathbb{R}\}$. It has a smallest element if $U = \mathbb{N}$.
- Divisibility: $(\mathbb{N}, |)$ is an order. 1 is the smallest element and 0 the greatest element. Greatest lower bound and least upper bound exist and are called greatest common divisor and least common multiple.
- For any order (A, r), the inverse $(A, (u,v) \mapsto r(v,u))$ is an order, too. This yields $(U, \geq)$ for numbers.
- For any order (A, r), the irreflexive variant (A, r') where $r'(u,v)$ holds if $r(u,v)$ and $u \neq v$ is still transitive. This yields $(U, <)$ and $(U, >)$ for numbers.
- For any preorder $(A, \prec)$, we obtain an equivalence relation $(A, \equiv)$ by putting $u \equiv v$ iff $u \prec v$ and $v \prec u$. In that case, we obtain an order $(A/\equiv, \prec)$ where $\prec$ is defined representative-wise: $[u]^{\equiv} \prec [v]^{\equiv}$ iff $u \prec v$.
- Inclusion of sets: $(\mathcal{P}(S), \subseteq)$ is a lattice. It has $\varnothing$ and S as the smallest and greatest element, respectively. $\cup$ and $\cap$ yield the least upper and the greatest lower bound of two sets, respectively.
- The diagonal relation: (A, Δ_A) where $\Delta_A(u,v)$ iff $u = v$ is a order and an equivalence relation.

- Equivalences: For any function $f : A \to B$, the model (A, r) with $r(u, v) = 1$ iff $f(u) = f(v)$ is an equivalence relation (and every equivalence relation is of this form).

Remark 3.50 (Uniqueness). For an antisymmetric relation, we have the following uniqueness properties: The least element is uniquely determined if it exists, and accordingly for the greatest element. Similarly, the least upper bound and the greatest lower bound of two elements is determined uniquely if it exists.

Remark 3.51 (Duality). By flipping the arguments of a binary predicate $\leq$, we obtain its dual $\geq$, i.e., $s \geq t$ is the same as $t \leq s$. If $\leq$ is symmetric, than $\leq$ and $\geq$ are interchangable.

We have the following correspondences:

$\leq$ is/has ...	iff	$\geq$ is/has ...
symmetric		symmetric
reflexive		reflexive
antisymmetric		antisymmetric
transitive		transitive
total		total
dense		dense
a *least* element e		a *greatest* element e
a *least* upper bound e of x and y		a *greatest* lower bound e of x and y

Remark 3.52 (Semilattices). Semilattices occur both in Ex. 3.1 and 3.3. This is because there are alternative but equivalent ways to see them.

Rem. 3.51 shows us that meet- and join-semilattices are dual, i.e., they are equivalent except for flipping the predicate. Therefore, we often just speak of semilattices.

Given a semilattice with function symbol $\circ$, we can define a binary predicate by $x \leq y$ iff $x \circ y \doteq x$. Then we can show that $\leq$ is a meet-semilattice. Conversely, given a meet-semilattice with predicate symbol $\leq$, we can define a binary function by putting $x \circ y$ to be the greatest lower bound of x and y. (This is well-defined because the greatest lower bound is uniquely determined by Rem. 3.50.) Then we can show that $\circ$ is a semilattice.

The according equivalence holds between semilattices and join-semi-lattices. Given a semilattice, the join-semilattice predicate is defined by $x \leq y$ iff $x \circ y \doteq y$. Conversely, given a join-semilattice, $x \circ y$ is defined as the least upper bound of x and y.

Moreover, the following are equivalent:

e is a ... element in the ...		
semilattice	join-semilattice	meet-semilattice
neutral	least	greatest
absorbing	greatest	least

Remark 3.53 (Lattices). Lattices also occur both in Ex. 3.2 and 3.3. By combining the constructions from Rem. 3.52, we construct the equivalence.

Consider a lattice in the sense of Ex. 3.2. In lattices, the two binary function symbols are usually written as $\sqcup$ instead of $\circ$ and $\sqcap$ instead of $*$. We define the predicate symbol by $x \leq y$ iff $x \sqcap y \doteq x$ or equivalently iff $x \sqcup y \doteq y$. Then we can show that $\leq$ is a lattice in the sense of Ex. 3.3.

Conversely, consider a lattice in the sense of Ex. 3.3. We define $x \sqcup y$ as the least upper bound and $x \sqcap y$ as the greatest lower bound of x and y. Then we can show that $\sqcup$ and $\sqcap$ yield a lattice in the sense of Ex. 3.2.

The equivalence extends to bounded lattices accordingly.

Data Types with Families of Unary Operators

Vector Spaces One of the most important data types of mathematics is vector spaces. However, each vector space requires two sets: a set V of vectors and a set F of scalars. Therefore, we also speak of F−vector spaces. Some of the operations relate the two sets, in particular the scalar multiplication $F \times V \to V$.

But $\mathcal{FOL}$-models V provide only one set $\texttt{term}^V$. Therefore, to specify vector spaces, we have several options:

- Use models whose universe is the set $F \cup V$. That is extremely awkward and should be avoided.
- Use a more expressive logic that allows for multiple sets. Such a logic exists: It is called $\mathcal{SFOL}$ for *sorted* first-order logic. Almost all definitions and results of $\mathcal{FOL}$ can be easily generalized to $\mathcal{SFOL}$.
- Assume that F is fixed and consider only V as the universe. This is a bit awkward and only works in a few cases. But in those few cases, it works well, and vector spaces are among them.

We follow the last option. So fix a field $F = (\texttt{term}^F, +^F, 0^F, -^F, 1^F, inv^F)$. The theory of F-vector spaces contains

- the symbols and axioms of commutative groups written as $+, 0, -$,
- for every $k \in \texttt{term}^F$ a unary function symbol m_k, and we write kv for $m_k(v)$,
- for every $k \in \texttt{term}^F$ one axiom for every function symbol of vector spaces, namely
  - $\forall v.\forall w.\ k(v + w) \doteq kv + kw$,
  - $k0 \doteq 0$,
  - $\forall v.\ k(-v) \doteq -(kv)$,
- for every symbol of the signature of fields a family of axioms, namely
  - for all $k, l \in \texttt{term}^F$, the axiom $\forall v.\ (k +^F l)v \doteq kv + lv$,
  - the axiom $\forall v.\ 0^F v \doteq 0$,
  - for all $k \in \texttt{term}^F$, the axiom $\forall v.\ (-^F k)v \doteq -(kv)$,
  - for all $k, l \in \texttt{term}^F$, the axiom $\forall v.\ (k \cdot^F l)v \doteq k(lv)$,
  - the axiom $1^F v \doteq v$,

Note that this theory is infinite if $\texttt{term}^F$ is. That is no problem in practice.

Above we have systematically included the superscript ^{F} on all field operations. That is usually omitted for simplicity.

The above list of axioms is some minor redundancies. But it is more systematic this way: Note that each axioms regulates what happens when a field operation "meets" a commutative group operation.

Finite Automata Just like vector spaces, finite automata require two sets: the set Q of states and the input alphabet I. We specify them as a $\mathcal{FOLEQ}$-theory using the same trick.

Fix an input alphabet I. The theories of deterministic and non-deterministic finite automata over I contain:

- a nullary function symbol *start*,
- a unary predicate symbols *final*,
- for every input symbol $x \in I$
  - in the deterministic case: a unary function symbol δ_x,
  - in the non-deterministic case: a binary predicate symbol δ_x.

For a deterministic automaton A, $\delta_x^A(q) = q'$ represent the transition from q to q' under input x. For a non-deterministic automaton A, $\delta_x^A(q, q') = 1$ represents a possible transition from q to q' under input x.

3.6.3 Negative Examples

It is one of the most unsatisfactory results about FOLEQ that a lot of interesting data types cannot be specified.

Example 3.54 (Natural Numbers). Consider the theory $(\Sigma; \Theta)$ for Peano arithmetic from Ex. 1.36. The **standard natural numbers** are the model

$$SN = (\mathbb{N}, 0, x \mapsto x + 1, +, \cdot)$$

We cannot specify the model class $\{SN\}$, i.e., we cannot give a theory Θ whose only model is $\{SN\}$. This would have be a theory whose theorems are

$$SN^* = \{F \in \mathbf{Sen}(\Sigma) \,|\, [\![F]\!]^{SN} = 1\}$$

This is no surprise: There are lots of other models that are isomorphic to SN and thus satisfy the same sentences. So the best we can hope for is to specify the class $\{I \in \mathbf{Mod}(\Sigma) \,|\, I \text{ isomorphic to } SN\}$.

However, not even that works: There are models of Peano arithmetic that are not isomorphic to SN but satisfy exactly the same formulas as SN. The problem is that their difference to SN cannot be expressed in FOLEQ. These are called non-standard models. Consequently, every theory that has SN as a model also has all the non-standard models.

It is hard to visualize non-standard models of arithmetic (see http://en.wikipedia.org/wiki/Non-standard_arithmetic). All of them are total orders that start with the natural numbers and are then followed by densely ordered copies of the integers. There are also non-standard models with uncountable universes.

Thus, the best we can do is to axiomatize the class

$$\mathcal{M} = \{I \in \mathbf{Mod}(\Sigma) \mid [\![F]\!]^I = [\![F]\!]^{SN} \text{ for all } F \in \mathbf{Sen}(\Sigma)\}$$

i.e., the class of all models satisfying the same sentences as SN.

But even that is difficult. We can try to axiomatize $\mathcal{M}$ using the axioms Θ of Peano arithmetic. But that is not enough: There are FOLEQ-sentences that hold in the natural numbers, but that are not consequences of Peano arithmetic, e.g., Goodstein's theorem (see http://en.wikipedia.org/wiki/Goodstein%27s_theorem). In particular, Peano arithmetic is not a complete theory.

Essentially the best theory we can find to axiomatize $\mathcal{M}$ is to use SN^* itself as the set of axioms. But that is like giving up: Then all the sentences that we would like to prove are axioms to begin with.

But even that is not a good solution. SN^* is not only infinite but – as we will see in Sect. 5.4 – not even recursively enumerable. Thus, we cannot even write down all the axioms in SN^* or tell which formulas are in it.

However, the theory of Peano arithmetic without multiplication does specify the model class $\{(\mathbb{N}, 0, x \mapsto x + 1, +)\}^{**}$. In particular, Peano arithmetic without multiplication is a complete theory.

Example 3.55 (Real Numbers). Similarly to Ex. 3.54, the real numbers cannot be specified in $\mathcal{FOLEQ}$.

Example 3.56 (Specification in Second-Order Logic). Both the natural and the real numbers can be specified in second-order logic (which we have not covered). Second-order logic permits quantifying over subsets of the universe. Using that, we can talk about induction for the natural numbers and, e.g., Dedekind cuts for the real numbers.

Example 3.57 (Specifying Universe Sizes). For a given signature,

- the class of all models with a universe of fixed finite size (i.e., all models with n elements for fixed n) *can* be specified in $\mathcal{FOLEQ}$ (Exercise!),
- the class of all models with a universe of fixed infinite size (e.g., all models with countably many elements) *cannot* be specified in $\mathcal{FOLEQ}$,
- the class of all models with finite universe *cannot* be specified in $\mathcal{FOLEQ}$,
- the class of all models with infinite universe *can* be specified in $\mathcal{FOLEQ}$ (Exercise!)

Chapter 4

Proof-Theoretic Semantics

4.1 Introduction

There is a number of philosophical arguments against model theory.

- In order to define consequence, mathematics (i.e., the collection of models) must already exist.
- The semantics of formulas is kind of trivial by interpreting, e.g., $\wedge$ as "and". Are we really formalizing anything, or are we only using fancy symbols?
- The model-theoretic definition of consequence is not accessible to algorithmic treatment.
- Model-theoretic consequence does not correspond to the everyday mathematical practice of proving theorems.
- Gödel's incompleteness shows: Consistency is a big problem.

In the following, we will introduce an inference system that forms the core of *proof theory.*
The most important example of a rule coming up in proof theory is the modus ponens rule

$$\dfrac{\vdash G}{\vdash F \to G \qquad \vdash F} R_2$$

Here $\vdash F$ is the judgment that F is true.
Further examples are this rule for conjunction and truth:

$$\dfrac{\vdash F \wedge G}{\vdash F \qquad \vdash G} R_1$$

$$\dfrac{\vdash \mathit{true}}{\ } R_3$$

This is the axiom of the so-called excluded middle or tertium non datur:

$$\dfrac{\vdash F \vee \neg F}{\ } R_4$$

Then proofs are the derivations of this inference system. For example, a proof of G using hypotheses from $\Delta = \{F_1, F_2, (F_1 \wedge (\mathit{true} \wedge F_2)) \to G\}$ could look like this:

$$\dfrac{\vdash G}{\dfrac{\vdash F_1 \wedge (\mathit{true} \wedge F_2)}{\vdash F_1 \qquad \dfrac{\vdash \mathit{true} \wedge F_2}{\dfrac{\vdash \mathit{true}}{\ } R_3 \qquad \vdash F_2} R_1} R_1 \qquad \vdash (F_1 \wedge (\mathit{true} \wedge F_2)) \to G} R_2$$

Here we apply R_3 to get $\vdash$ *true*, then R_1 twice to get $\vdash F_1 \wedge (\mathit{true} \wedge F_2)$, then R_2 to get G. Note how the leaves of the derivation tree are either axioms (i.e., $\vdash$ *true*) or hypotheses (i.e., $\vdash F$ for $F \in \Delta$). We can think of the hypotheses as additional axioms.

An inference system used for proof theory is often also called a *calculus*. In fact, first-order logic is often called (first-order) predicate calculus. Calculus-based definitions of consequence have important properties that answer the criticisms against model theory:

- It is algorithmic: Given a calculus, we can
  - search for proofs,
  - check whether a given proof is correct.
- It is defined by giving rules that directly capture our intuition.
- No model theory and almost nothing about mathematics is presupposed.
- It is very similar to how a mathematician would reason.

Now the crucial design question is: How do we get a calculus for PL, FOL, FOLEQ?

- First design decision: What are the judgments?
  - simplest choice: the Σ-judgments are $\vdash F$ for Σ-formulas F
  - more complicated choices of judgments are much more useful in practice
- Second design decision: What are the rules?
  - Very different sets of rules may lead to the same consequence relation
  - Very similar sets of rules may lead to very different implementations
- Conflicting design goals:
  - Theoretical elegance, simplicity
  - Practical strength, efficiency — We want to use calculi to make the computer find proofs!

4.2 Calculi for Propositional Logic

4.2.1 Hilbert-Calculi

The simplest calculi are those where the judgments are simply the formulas. Such calculi are called Hilbert calculi. Assume a PL-signature Σ. A calculus $C(\Sigma)$ is given by using the set $\mathit{Form}(\Sigma)$ as the set of judgments and the following rules:

- modus ponens

$$\frac{G}{F \to G \quad F}MP$$

- the following axioms (i.e.) rules without hypotheses (where we use the $\leftrightarrow$ as an abbreviation)

  1. $\mathit{false} \leftrightarrow \neg \mathit{true}$
  2. $(F \to G) \leftrightarrow (\neg F \vee G)$
  3. $(F \wedge G) \leftrightarrow \neg(\neg F \vee \neg G)$
  4. *true*
  5. $(F \vee F) \to F$
  6. $F \to (F \vee G)$
  7. $(F \vee G) \to (G \vee F)$

8. $(G \to H) \to ((F \lor G) \to (F \lor H))$

All these rules are used for arbitrary formulas F and G. Therefore, we actually have infinitely many rules.

This calculus is essentially the one used in the Bertrand's and Russell's Principia [?], a very influential work from 1910 that attempted to formalize all of mathematics. For example, Gödel discovered his result [?] while working in the Principia, i.e., they were the mathematics as he knew it.

An overview over Hilbert Calculi for various variants of propositional logics is given at http://home.utah.edu/~nahaj/logic/structures/systems/index.html

Assume $\Sigma = \{p, q\}$. Then an example proof with hypothesis $\Delta = \{p\}$ looks like this:

$$\dfrac{\dfrac{}{(p \lor q) \to (q \lor p)}\,\mathit{Axiom}\ 7 \qquad \dfrac{\dfrac{}{p \to (p \lor q)}\,\mathit{Axiom}\ 6 \qquad \dfrac{}{p}\,\Delta}{p \lor q}\,MP}{q \lor p}\,MP$$

Hilbert calculi are extremely useful in theory. Because there is only one rule, the proofs are very simple: The leaves are axioms, and then only modus ponens is used.

For the same reason, Hilbert calculi are extremely useless in practice. For example, try to derive something as trivial as $F \to F$.

4.2.2 Other Calculi

Other calculi for PL are obtained by dropping the rules for the quantifiers from the following calculi for FOL.

4.3 Calculi for First-Order Logic

4.3.1 The Natural Deduction Calculus

The point of natural deduction is that it captures the way in which we reason intuitively and mathematically.

The main judgment of the ND Calculus is

$$\Gamma; \Delta \vdash_\Sigma F$$

where Γ is a Σ-context, Δ is a list of Σ-formulas in context Γ and F is a Σ-formula in context Γ. The rules of the ND calculus are described in detail below.

Notation 4.1. A lot of simplifications are common:

- The subscript Σ in $\vdash_\Sigma$ is often dropped when Σ is fixed.
- We usually write $x_1, \dots, x_m$ instead of $\{x_1, \dots, x_m\}$ and Γ, x instead of $\Gamma \cup \{x\}$.
- Similarly, we usually write $F_1, \dots, F_m$ instead of $\{F_1, \dots, F_m\}$ and Δ, F instead of $\Delta \cup \{F\}$.
- In $\Gamma; \Delta \vdash F$, we omit Γ or Δ if they are empty.
- In $\Gamma; \Delta \vdash F$, usually Γ is omitted altogether. In that case, it is implied that Γ is the set of all variables occurring free in any formula in Δ or in F.

The intuition of the judgment $\{x_1, \dots, x_m\}; \{F_1, \dots, F_n\} \vdash F$ is as follows: For arbitrary values for $x_1, \dots, x_m$, if all of the $F_1, \dots, F_n$ hold, then F holds. Using this intuition, the rules of ND capture the meaning we have in mind when we prove something. That's why it is called *natural* deduction.

The rules are split into elimination and introduction rules. We use introduction rules when we establish a proof goal. For example, we use $\land I$ to establish the *proof goal* $\vdash A \land B$ after first proving $\vdash A$ and $\vdash B$. We use elimination rules when we use assumptions. For example, we use $\land E_l$ to break down the previously proved $\vdash A \land B$ into $\vdash A$.

When we actually do a proof in this calculus, we typically apply introduction rules backwards (i.e., from bottom to top) as long as possible. Then we use elimination rules forwards (i.e., from top to bottom) to build up the needed proof goal from the assumptions.

Fig. 4.1 gives the introduction and elimination rules for ND as well as some structural rules. All rules are given for arbitrary Γ, Δ, A, and B. To indicate that Σ, Γ, and Δ are irrelevant (and could be omitted), they are printed in gray.

	Introduction	Elimination
$\wedge$	$\dfrac{\Gamma;\Delta\vdash_\Sigma A\wedge B}{\Gamma;\Delta\vdash_\Sigma A \quad \Gamma;\Delta\vdash_\Sigma B}\wedge I$	$\dfrac{\Gamma;\Delta\vdash_\Sigma A}{\Gamma;\Delta\vdash_\Sigma A\wedge B}\wedge E_l \qquad \dfrac{\Gamma;\Delta\vdash_\Sigma B}{\Gamma;\Delta\vdash_\Sigma A\wedge B}\wedge E_r$
$\vee$	$\dfrac{\Gamma;\Delta\vdash_\Sigma A\vee B}{\Gamma;\Delta\vdash_\Sigma A}\vee I_l \qquad \dfrac{\Gamma;\Delta\vdash_\Sigma A\vee B}{\Gamma;\Delta\vdash_\Sigma B}\vee I_r$	$\dfrac{\Gamma;\Delta\vdash_\Sigma C}{\Gamma;\Delta\vdash_\Sigma A\vee B \quad \Gamma;\Delta,A\vdash_\Sigma C \quad \Gamma;\Delta,B\vdash_\Sigma C}\vee E$
$\rightarrow$	$\dfrac{\Gamma;\Delta\vdash_\Sigma A\rightarrow B}{\Gamma;\Delta,A\vdash_\Sigma B}\rightarrow I$	$\dfrac{\Gamma;\Delta\vdash_\Sigma B}{\Gamma;\Delta\vdash_\Sigma A\rightarrow B \quad \Gamma;\Delta\vdash_\Sigma A}\rightarrow E$
$\neg$	$\dfrac{\Gamma;\Delta\vdash_\Sigma \neg A}{\Gamma;\Delta,A\vdash_\Sigma \mathit{false}}\neg I$	$\dfrac{\Gamma;\Delta\vdash_\Sigma C}{\Gamma;\Delta\vdash_\Sigma \neg A \quad \Gamma;\Delta\vdash_\Sigma A \quad \Gamma\vdash_\Sigma C:\texttt{form}}\neg E$
true	$\dfrac{\Gamma;\Delta\vdash_\Sigma \mathit{true}}{}\mathit{trueI}$	
false		$\dfrac{\Gamma;\Delta\vdash_\Sigma C}{\Gamma;\Delta\vdash_\Sigma \mathit{false} \quad \Gamma\vdash_\Sigma C:\texttt{form}}\mathit{falseE}$
$\forall$	$\dfrac{\Gamma;\Delta\vdash_\Sigma \forall x\ A}{\Gamma,x;\Delta\vdash_\Sigma A \quad x\notin\Gamma}\forall I$	$\dfrac{\Gamma;\Delta\vdash_\Sigma A[x/t]}{\Gamma;\Delta\vdash_\Sigma \forall x\ A \quad \Gamma\vdash_\Sigma t:\texttt{term}}\forall E$
$\exists$	$\dfrac{\Gamma;\Delta\vdash_\Sigma \exists x\ A}{\Gamma;\Delta\vdash_\Sigma A[x/t] \quad \Gamma\vdash_\Sigma t:\texttt{term}}\exists I$	$\dfrac{\Gamma;\Delta\vdash_\Sigma C}{\Gamma;\Delta\vdash_\Sigma \exists x\ A \quad \Gamma,x;\Delta,A\vdash_\Sigma C \quad x\notin\Gamma,C}\exists E$
	$\dfrac{\Gamma;\Delta\vdash A}{A\in\Delta}\mathit{Axiom} \qquad \dfrac{\Gamma;\Delta\vdash_\Sigma A\vee\neg A}{}\mathit{tnd}$	$\dfrac{\Gamma;\Delta\vdash_\Sigma B}{\Gamma;\Delta\vdash_\Sigma A \quad \Gamma;\Delta,A\vdash_\Sigma B}\mathit{cut}$

Figure 4.1: Natural Deduction Rules

The intuitions behind the rules are as follows.

- $\wedge I$, $\wedge E_l$, $\wedge E_r$: Proving a conjunction is equivalent to proving the two conjuncts.
- $\vee I_l$, $\vee I_r$: To prove a disjunction, it suffices to prove one of the disjuncts.
- $\vee E$: If a disjunction has been proved, it can be used for case distinction on the two disjuncts.
- $\rightarrow I$: Proving an implication means to assume the implicant and to prove the implicate.
- $\rightarrow E$: This is modus ponens.
- $\neg I$: Proving a negation means to prove that the opposite leads to a contradiction. This is also called indirect reasoning. There are various ways to express contradictions; here we use proving $\neg A$ from A; another option is to prove *false* or to prove all formulas (i.e., to prove an arbitrary formula).
- $\neg E$: From a contradiction (i.e., A and $\neg A$ provable), we can prove everything. Compare *falseE*.
- *trueI*: *true* always holds.
- *falseE*: *false* never holds, i.e., if it holds, then there is a contradiction and every well-formed formula holds. Note that there is an additional assumption that C is a well-formed formula. This is necessary because otherwise C could be any nonsensical object.

- $\forall I$: A universal quantification holds, if its body holds for an arbitrary value. The "arbitrary value" is represented by a fresh variable, i.e., a variable that occurs nowhere.
- $\forall E$: If a universal quantification holds, its body can be instantiated with every well-formed term.
- $\exists I$: An existential quantification holds, if its body holds for some value. The "some value" is represented by an arbitrary well-formd term.
- $\exists E$: If an existential quantification holds, then its body must holds for some value. The "some value" is represented by a fresh variable.
- *Axiom*: The assumptions imply themselves.
- *tnd*: Tertium non datur, either a formula or its negation holds.
- *cut*: This rule is lemma application, i.e., if a formula is proved separately, then it can be used as an additional assumption.

An important question in a given calculus is the redundancy of rules, especially of the structural rules, which are often not so natural. A rule is redundant if omitting the rule does not change the set of derivable judgments and thus does not change the consequence relation. We have the following:

Remark 4.2 (Tertium Non Datur). The rule *tnd* is not redundant. On the other hand, the system of introduction and elimination rules has a nice symmetry, and adding *tnd* to it seems arbitrary and unnatural. If we omit it, some formulas are not provable anymore. For example, there would be no proofs for $\neg\neg A \to A$ and $\neg\forall x\ F \to \exists x\ \neg F$, which hold in every FOL model. However, some philosophers and researchers argue that would be a feature, not a bug, and that the FOL model theory is wrong, not the proof theory. This point of view is called *intuitionism* and has led to the development of *intuitionistic logic*, which does not use *tnd*.

Remark 4.3 (Cut). The rule *cut* is redundant. To prove that it is redundant for a given logic, is called *cut* elimination. *cut* elimination is one of the central problems in proof theory because once *cut* is eliminated, it is relatively easy to show consistency. (An inference system is consistent, if not every formula is derivable. An inconsistent inference system is clearly useless.) Also proof search is a lot easier for a machine (For a human, it is harder.) if *cut* is omitted.

Remark 4.4 (Empty Universe). As pointed out in Rem. 3.16, most textbooks exclude the case $\texttt{term}^I = \varnothing$. Those textbooks use subtly different and slightly stronger rules for the quantifiers, which have the side-effect that they prove $\exists x.true$. Thus, these textbook definitions of $\mathcal{FOLEQ}$ work out in the sense that both the model theory and the proof theory exclude the empty universe.

Our calculus does not prove $\exists x.true$. Therefore our model and proof theory work out in the sense that both allow the empty universe.

If we want to use our calculus for the variant of the model theory where the empty model is excluded, we have to add an appropriate axiom, e.g., $\exists x.true$.

Rules for Equality So far we have only considered FOL. By adding the rules from Fig. 4.2, we obtain the natural deduction calculus for FOLEQ.

The intuitions behind the rules are as follows.

- $refl$: Every term is equal to itself (reflexivity).
- *sym*: Equality is symmetric.
- *trans*: Equality is transitive.
- *congterm*: Every function $U(x)$ maps equal inputs s and t to equal outputs $U(s)$ and $U(t)$ (congruence for terms).
- *congform*: If a property $U(x)$ holds for s then it also holds for anything equal to s (congruence for formulas).

The first three rules have the effect that equality is an equivalence relation. *congterm* makes it additionally a congruence relation. Finally, *congform* say that formulas can never distinguish between equal terms.

The rules *sym* and *trans* are actually redundant: They can be proved using $refl$ and *congform* (Exercise!).

$$\frac{\Gamma \vdash_\Sigma t : \texttt{term}}{\Gamma; \Delta \vdash_\Sigma t \doteq t}\,refl \qquad \frac{\Gamma; \Delta \vdash_\Sigma s \doteq t}{\Gamma; \Delta \vdash_\Sigma t \doteq s}\,sym \qquad \frac{\Gamma; \Delta \vdash_\Sigma r \doteq s \quad \Gamma; \Delta \vdash_\Sigma s \doteq t}{\Gamma; \Delta \vdash_\Sigma r \doteq t}\,trans$$

$$\frac{\Gamma; \Delta \vdash_\Sigma s \doteq t \quad \Gamma, x \vdash_\Sigma U : \texttt{term}}{\Gamma; \Delta \vdash_\Sigma U[x/s] \doteq U[x/t]}\,congterm$$

$$\frac{\Gamma; \Delta \vdash_\Sigma s \doteq t \quad \Gamma, x \vdash_\Sigma U : \texttt{form} \quad \Gamma; \Delta \vdash_\Sigma U[x/s]}{\Gamma; \Delta \vdash_\Sigma U[x/t]}\,congform$$

Figure 4.2: Equality Rules

The Inference System Finally, we can define the inference system of natural deduction.

Definition 4.5. The inference system $\mathbf{Pf}^{FOLEQ}(\Sigma)$ consists of the following **judgments**

- all judgments $\Gamma \vdash_\Sigma t : \texttt{term}$ for any Γ, t,
- all judgments $\Gamma \vdash_\Sigma F : \texttt{form}$ for any Γ, F,
- all judgments $\Gamma; \Theta \vdash_\Sigma F$ for any Γ, Θ, F.

and the proof rules of Fig. 4.1 and 4.2.

The Deduction Theorem Interestingly, we now have two options to define provability:

Definition 4.6. We say that $F \in \mathbf{Sen}^{FOLEQ}(\Sigma)$ is **locally provable** from assumptions $F_1, \ldots, F_n \subseteq \mathbf{Sen}^{FOLEQ}(\Sigma)$ if there is a proof

$$\frac{F_1, \ldots, F_n \vdash_\Sigma F}{}$$

We say that $F \in \mathbf{Sen}^{FOLEQ}(\Sigma)$ is **globally provable** from assumptions $\Theta \subseteq \mathbf{Sen}^{FOLEQ}(\Sigma)$ if there is a proof

$$\frac{\vdash_\Sigma F}{\vdash_\Sigma F_1 \quad \ldots \quad \vdash_\Sigma F_n}$$

Global provability is a very general notion because it can be stated for virtually any calculus, e.g., for a Hilbert calculus. Local provability depends on the structure of the ND calculus: We can only state it because ND judgments may have assumptions, e.g., we cannot state if for a Hilbert calculus.

It turns out for the ND calculus, both notions are equivalent:

Theorem 4.7 (Deduction Theorem)**.** *In the ND calculus for FOLEQ, local and global provability are equivalent.*

Proof. Firstly, assume a provability $F_1, \ldots, F_n \vdash_\Sigma F$. By applying $\to I$ n times, we obtain a proof of $\vdash_\Sigma F_1 \to \ldots \to F_n \to F$. By applying $\to E$ n times, we obtain global provability. The opposite direction proceeds accordingly. □

4.4 An Abstract Definition of the Proof Theory of a Logic

In Sect. 1.4 and 3.4, we have seen abstract definitions of syntax ($\mathbf{Sig}, \mathbf{Sen}$) and for a given logic syntax the model theory ($\mathbf{Mod}, \models$) of logics. We will now the analogue for proof theory: for a given logic syntax, define the proof theory of a logic.

Definition 4.8 (Proof Theory)**.** A proof theory for the logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ is a pair $(\mathbf{Pf}, \vdash)$ such that for every $\Sigma \in \mathbf{Sig}$, we have that

- $\mathbf{Pf}(\Sigma)$ is an inference system,
- for any sentences $\Theta \subseteq \mathbf{Sen}(\Sigma)$ and $F \in \mathbf{Sen}(\Sigma)$, there is a judgment of $\Theta \vdash F$ in $\mathbf{Pf}(\Sigma)$.

If there is a proof $p \in \mathbf{Pf}(\Sigma)$ of $\Theta \vdash F$ in $\mathbf{Pf}(\Sigma)$, we say F is provable from/implied by/entailed by/a consequence of Θ.
A four-tuple $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash)$ of a logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ and a proof theory $(\mathbf{Pf}, \vdash)$ for it is called a *proof theoretical logic*.

Example 4.9. We immediately get some examples:

- We have a proof theoretical logic $FOLEQ = (\mathbf{Sig}^{FOLEQ}, \mathbf{Sen}^{FOLEQ}, \mathbf{Pf}^{FOLEQ}, \vdash^{FOLEQ})$. $\mathbf{Pf}^{FOLEQ}$ is defined in Def. 4.5. $\Theta \vdash^{FOLEQ}_{\Sigma} F$ is already defined as an abbreviation for $\varnothing; \Theta \vdash_{\Sigma} F$.
- Proof theoretical logics for FOL, ALGEQ, and HORNEQ are obtained from FOLEQ in the obvious way.

4.5 Theorems and Consequence (Proof-Theoretically)

In Sect. 3.5, we defined model theoretical theorems and consequence for an arbitrary model theory. Now we do the same proof theoretically for an arbitrary proof theory. We obtain the proof theoretical analogues to Def. 3.26, Def. 3.31, Def. 3.34, Lem. 3.35, and 3.36:

Definition 4.10 (Theorems)**.** Given a proof theoretical logic $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash)$ and a theory $(\Sigma; \Theta)$. Then a Σ-sentence is a (proof-theoretical) *theorem/tautology/consequence/valid sentence* of $(\Sigma; \Theta)$ if there is a proof of

$$\Theta \vdash_{\Sigma} F$$

and we write

$$Thm(\Sigma; \Theta) = \{F \in \mathbf{Sen}(\Sigma) \mid \Theta \vdash_{\Sigma} F\}$$

F is called a *contradiction* if it $\Theta, F \vdash_{\Sigma} C$ for all $C \in \mathbf{Sen}(\Sigma)$.

Lemma 4.11. *If a proof theoretical logic contains a negation whose rules are as for the ND calculus: F is a theorem iff $\neg F$ is a contradiction. F is a contradiction iff $\neg F$ is a theorem.*

Proof. Clear using the rules $\neg I$ and $\neg E$. □

Definition 4.12. A theory $(\Sigma; \Theta)$ is called

- *inconsistent* if all sentences are theorems, $\Theta \vdash_{\Sigma} F$ for every $F \in \mathbf{Sen}(\Sigma)$,
- *consistent* if it is not inconsistent, i.e., there is a sentence that is not a theorem,
- *complete* if every sentence is either a theorem or a contradiction, i.e., if for every $F \in \mathbf{Sen}(\Sigma)$ either $\Theta \vdash_{\Sigma} F$ or $\Theta \vdash_{\Sigma} \neg F$.

Lemma 4.13. *The following are equivalent for a theory $(\Sigma; \Theta)$ in any proof theoretical logic L:*

1. *It is inconsistent.*
2. *(If L has negation in the same way as FOL:) F and $\neg F$ are theorems for some F.*

3. *(If L has falsity in the same way as FOL:) false is a theorem.*

Proof. Easy using the rules for negation and falsity. □

Notation 4.14. We have defined the notions "consequence", "theorem", "contradiction", "(in)consistent", and "complete" proof-theoretically. All have model-theoretical analogues, see Def. 3.26, 3.31, and 3.34. If we need to distinguish them, we will prefix P, e.g., we will say that a theory is P-consistent. See also Not. 3.37.

Chapter 5

The Relation between Proof and Model Theory

5.1 An Abstract Definition of a Logic

Definition 5.1 (Logic)**.** A *logic* consists of

- a collection of signatures $\mathbf{Sig}$,
- a set of sentences $\mathbf{Sen}(\Sigma)$ for every $\Sigma \in \mathbf{Sig}$,
- a collection of models $\mathbf{Mod}(\Sigma)$ for every $\Sigma \in \mathbf{Sig}$,
- a model-theoretic definition of truth and consequence: for every $\Sigma \in \mathbf{Sig}$, a relation $\models_\Sigma$ between $\mathbf{Mod}(\Sigma)$ and $\mathbf{Sen}(\Sigma)$,
- an inference system $\mathbf{Pf}\,(\Sigma)$ for every $\Sigma \in \mathbf{Sig}$,
- a proof theoretic definition of consequence: for every $\Sigma \in \mathbf{Sig}$, a judgment $\Delta \vdash_\Sigma F$.

Thus, we have two alternative ways of defining theorems and consequence: model and proof theoretically. The former was strongly influenced by Tarski [**?**] and [**?**] and has become the dominant method in mathematical logic. The latter was strongly influenced by Hilbert's program [**?**] and the work by Gödel [**?**] and Gentzen [**?**]. It has become important in computer science because it permits using computers to check and search for proofs.

5.2 Soundness and Completeness

Proof and model theory work best when used in combination: We use theories as interfaces, models as interface providers, and the proofs of a theory can be reused in every model. This works if P-consequence and M-consequence are the same, which is where soundness and completeness come in.

5.2.1 Definitions

We define:

Definition 5.2 (Soundness and Completeness)**.** A *logic* is *sound* if for all signatures Σ, all sets of sentences Δ and all sentences F,

$$\Delta \vdash_\Sigma F \qquad \text{implies} \qquad \Delta \models_\Sigma F.$$

A *logic* is *complete* if for all signatures Σ, all sets of sentences Δ and all sentences F,

$$\Delta \models_\Sigma F \qquad \text{implies} \qquad \Delta \vdash_\Sigma F.$$

In other words, it is complete if M-consequence implies P-consequence; or if all M-theorems are P-theorems. And it is sound if the implication goes the other way around.

Notation 5.3. Note that both theories and logics can be complete. The two notions are only loosely related and should be seen as different notions that happen to share the name "complete".

Soundness and completeness extend to the other notions mentioned in Not. 3.37 and 4.14:

Theorem 5.4. *If a logic is sound then,*

- *P-contradictions are M-contradictions,*
- *P-inconsistent theories are M-inconsistent,*
- *P-complete theories are M-complete.*

For completeness, the opposite implications hold.

Proof. Straightforward because all notions are defined in terms of the consequence relation. □

Note that Thm. 5.4 does not include a case for "consistent" theories. For "consistent", the implication is flipped because "M-consistent theories are P-consistent" is the same as "not M-inconsistent theories are not P-inconsistent" and that is equivalent to "P-inconsistent theories are M-inconsistent". More precisely, we have the following:

5.2.2 Intuitions

Soundness means that the proof theory is correct: If we can prove a consequence, it holds about all models. It is usually easy to show soundness by induction on proof trees.

Completeness is more complicated (and there are useful logics that are not complete). To understand it better, we show the following:

Lemma 5.5. *The following are equivalent for any logic L that has a negation together with rules for negation introduction and double-negation elimination (i.e., $\neg\neg F \vdash F$):*

1. *L is complete (i.e., M-consequence implies P-consequence).*
2. *Every P-consistent theory has a model (i.e., P-consistency implies M-consistency).*

Proof. By circular implication:
1 implies 2: See Thm. 5.4. 2 implies 1: To show completeness, assume a theory $(\Sigma;\Theta)$ and an M-theorem F (*). Then $(\Sigma;\Theta\cup\{\neg F\})$ has no model. Using 2, we obtain that $(\Sigma;\Theta\cup\{\neg F\})$ is P-inconsistent. Using negation introduction, we can derive $\neg\neg F$ from Θ, and because of double-negation elimination, we can also derive F. Thus, F is a P-theorem of $(\Sigma;\Theta)$. □

Lem. 5.5 is the typical way how we prove completeness.

Then we have two intuitions for the word "completeness". If the model theory defines the theorems, completeness is a property of the proof theory: A logic is complete if it has enough proofs to derive all the theorems.

If the proof theory defines the theorems, completeness is a property of the model theory: A logic is complete if it has enough models to interpret all the consistent theories.

Another way to think about soundness and completeness is to imagine a step-by-step logic design. Say initially we have no proof rules and no models. Then no formula is a P-theorem, and all formulas are M-theorems. The logic is as sound as it can be but also as far from complete as it can be.

We can change the logic by adding proof rules (which increases the set of P-theorems) and/or by adding models (which decreases the set of M-theorems). If we have added enough proof rules and models, the sets of P-theorems and M-theorems meet, and we have a sound and complete logic. If we keep adding proof rules or models, we will have too many: The set of P-theorems becomes bigger than the set of M-theorems, and we lose soundness (while staying complete).

Thus, to make a logic sound, we have to remove rules and/or models. And to make a logic complete, we have to add rules and/or models.

Example 5.6 (Empty Universe). As pointed in Rem. 3.16 and 4.4, there is flexibility as to how to treat the empty universe.

If we use the usual rules for first-order logic with all models (including the empty universe), the logic is not sound. To make it sound, we have two options:

- remove models, specifically the one with the empty universe: This is the option chosen by most logic textbooks.
- remove (or weaken) rules: This is the option chosen in these notes by using weakening the $\forall E$ rule.

5.2.3 Compactness

Compactness is the property that all consequences are caused by finitely many axioms:

Definition 5.7. A model theory is called **compact** if $\Sigma \models_\Theta F$ implies $\Sigma \models_{\Theta'} F$ for some finite set $\Theta' \subseteq \Theta$.

Theorem 5.8. *If a logic is sound and complete, then it is compact.*

Proof. We always have that $\Theta \vdash_\Sigma F$ implies $\Theta' \vdash_\Sigma F$ for some finite set $\Theta' \subseteq \Theta$ because a proof can only use finitely many axioms from Θ.

Therefore, soundness and completeness together imply compactness. □

5.3 The Big Theorems of First-Order Logic

The most important theorems about FOL and FOLEQ are soundness and completeness. The first sound and complete calculi for first-order logic were given by Gödel [?] and Gentzen [?] in the 1930s.

5.3.1 Soundness

Theorem 5.9 (Soundness of FOL). *FOL is sound.*

Proof. By induction on derivations. The theorem only mentions those judgments where $\Gamma = \varnothing$ (i.e., Γ is omitted). This is as usual the only case we are really interested in. But in the induction we will encounter all judgments.

Therefore, we have to prove something more general, namely:

$\Gamma; \Delta \vdash_\Sigma F$ implies that
  for every model $I \in \mathbf{Mod}(\Sigma)$ and every assignment α for Γ,
  we have that
    if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$,
    then $I, \alpha \models_\Sigma F$.

Note that if we put $\Gamma = \varnothing$, this is the same as saying $\Delta \models_\Sigma F$.

Now we can prove this more general statement by induction on derivations. The intuition behind this is the following: For all axioms $\dfrac{\vdash A}{}$, show that A is true in every model. For every rule $\dfrac{J}{J_1 \quad \dots \quad J_n}$ show that applying the rule preserves truth, i.e., show the above for J assuming it is true (induction hypothesis) for the J_i. Since proof trees are built up by composing rules, the induction requires one case for every rule. Therefore, we often say that a single rule is sound: The soundness of a rule R means that the induction step succeeds for R; then to show soundness of the logic, we need to show that every rule is sound.

In the following we give three example soundness proofs for the rules *Axiom*, $\wedge I$, and $\rightarrow I$.

The case for *Axiom*. For this rule, there are no hypotheses. So we have to show that for every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$, then $I, \alpha \models_\Sigma A$. This is trivial because $A \in \Delta$.

The case for $\wedge I$. For this rule, there are two hypotheses. Applying the induction hypothesis to them yields:

- (1) For every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$, then $I, \alpha \models_\Sigma A$.
- (2) For every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$, then $I, \alpha \models_\Sigma B$.

We have to show: For every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$, then $I, \alpha \models_\Sigma A \wedge B$. This is obvious. But to be totally clear, here is the full proof:

- We pick an arbitrary model I_0 and an arbitrary assignment α_0.
- We assume that (3) $I_0, \alpha_0 \models_\Sigma T$ for all $T \in \Delta$. We have to show $I_0, \alpha_0 \models_\Sigma A \wedge B$.
  - From (3), using (1) and (2), we obtain (4) $I_0, \alpha_0 \models_\Sigma A$ and (5) $I_0, \alpha_0 \models_\Sigma B$, respectively.
  - From (4) and (5), using the definition of the interpretation of formulas, we obtain $I_0, \alpha_0 \models_\Sigma A \wedge B$.
  - Done.
- Done.

The case for $\rightarrow I$. For this rule, there is one hypothesis. Applying the induction hypothesis to it yields:

- (1) For every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta \cup \{A\}$, then $I, \alpha \models_\Sigma B$.

We have to show: For every model and assignment I and α, if $I, \alpha \models_\Sigma T$ for all $T \in \Delta$, then $I, \alpha \models_\Sigma A \rightarrow B$.

- We pick an arbitrary model I_0 and an arbitrary assignment α_0.
- We assume that (2) $I_0, \alpha_0 \models_\Sigma T$ for all $T \in \Delta$. We have to show $I_0, \alpha_0 \models_\Sigma A \rightarrow B$.
- We distinguish two cases: $I_0, \alpha_0 \models_\Sigma A$ and $I_0, \alpha_0 \not\models_\Sigma A$.
  - $I_0, \alpha_0 \not\models_\Sigma A$. Then by definition of the interpretation of formulas $I_0, \alpha_0 \not\models_\Sigma A \rightarrow B$. Done.
  - (3) $I_0, \alpha_0 \models_\Sigma A$.
    * Combining (2) and (3), we are able to apply (1), from which we obtain (4) $I_0, \alpha_0 \models_\Sigma B$.
    * From (3) and (4), using the definition of the interpretation of formulas, we obtain $I_0, \alpha_0 \models_\Sigma A \rightarrow B$.
    * Done.
- Done.

□

5.3.2 Completeness

The completeness of FOL requires a few additional constructions.
The central idea of the completeness proof is to build a model S of a theory $(\Sigma; \Theta)$ from the terms and proofs. First, we define the interpretation of function symbols – this is called the term model:

Definition 5.10 (Term Model)**.** Let Σ be a signature without predicate symbols.
Then we define the model S as follows:

- $\mathtt{term}^S = \{t \mid \vdash_\Sigma t : \mathtt{term}\}$
- $f^S(u_1, \ldots, u_n) = f(u_1, \ldots, u_n)$ for $n = ar(f)$

Remark 5.11. If we used the variant of $\mathcal{FOLEQ}$ that excludes the empty universe, we might have to add some nullary function symbol to Σ to make sure that the universe of the term model is not empty.

Read the last line in Def. 5.10 very carefully:

1. $\mathtt{term}^S$ is a set, which S defines to be set of terms.
2. f is a function symbol.
3. f^S is an n-ary function on $\mathtt{term}^S$, i.e., a function that takes terms and returns a term.
4. $f^S(u_1,\ldots,u_n)$ is the application of f^S to some arguments, which in this case must be terms.
5. $f(u_1,\ldots,u_n)$ is the term formed from the function symbol f and the term u_i.

The important property of the term model is that it interprets every term as itself:

Lemma 5.12. *In the situation of Def. 5.10, we have*

$$[\![t]\!]^S = t$$

for all terms $\vdash_\Sigma t : \mathtt{term}$.

Proof. We use induction on t:

- Case $t = f(t_1,\ldots,t_n)$: We have

$$[\![f(t_1,\ldots,t_n)]\!]^S =^{Def} f^S([\![t_1]\!]^S,\ldots,[\![t_n]\!]^S) =^{IH} f^S(t_1,\ldots,t_n) =^{Def} f(t_1,\ldots,t_n)$$

- Case $t = x$: impossible because t has no free variables.

□

You should read the proof very carefully, too, and understand how the three steps in the case for function terms proceed.
We cannot interpret every formula as itself because the set $\mathtt{form}^S$ is already fixed to be $\{0,1\}$. But we can do something similar: We interpret a formula F as 1 if it is a theorem and as 0 if is a contradiction. But that does not quite work immediately: If Θ is not complete, there are formulas that are neither theorems nor contradictions. Therefore, we show:

Lemma 5.13. *For every P-consistent theory Θ, there is a P-consistent complete theory $\overline{\Theta}$ with $\Theta \subseteq \overline{\Theta}$.*

Proof. Let $F_1, F_2, \ldots,$ be an enumeration of all Σ-sentences (in any order). We define theories $\Theta_0, \Theta_1, \Theta_2, \ldots$ inductively:

- $\Theta_0 = \Theta$
- $\Theta_{n+1} = \begin{cases} (a) & \Theta_n \cup \{F_{n+1}\} & \text{if not } \Theta_n \vdash_\Sigma \neg F_{n+1} \\ (b) & \Theta_n & \text{otherwise} \end{cases}$

We define $\overline{\Theta} = \bigcup_{i\in\mathbb{N}} \Theta_i$.

We have to show three properties about $\overline{\Theta}$.

- $\Theta \subseteq \overline{\Theta}$: That is obvious.
- Completeness: Consider a sentence F. We know $F = F_n$ for some n. By construction $\overline{\Theta}$ contains F_n for every sentence or can prove $\neg F_{n+1}$. Thus it can prove F_n or $\neg F_n$.
- P-consistency:

1. We show that if Θ_{n+1} is P-inconsistent, then so is Θ_n. Assume Θ_{n+1} is P-inconsistent. We distinguish the two cases in the definition of Θ_{n+1}:
   (a) Then we have $\Theta_n, F_{n+1} \vdash_\Sigma \mathit{false}$ and thus (using the negation introduction rule) $\Theta \vdash_\Sigma \neg F_{n+1}$. But that violates the assumption of case (a). There are two cases for G:
   (b) Then $\Theta_{n+1} = \Theta_n$, so Θ_n is also inconsistent.
2. Thus, because Θ_0 is P-consistent, so are all Θ_n.
3. If $\overline{\Theta}$ were P-inconsistent, there would be a proof p of $\overline{\Theta} \vdash_\Sigma \mathit{false}$. Because a proof can only use finitely many axioms, there would be a Θ_n that contains all axioms used in p. Then Θ_n would be inconsistent, too, which is impossible.

□

Then we can extend Def. 5.10:

Definition 5.14 (Term Model with Predicates)**.** For a theory $(\Sigma; \Theta)$, we define the model S as follows:

- $\texttt{term}^S$ and f^S are as in Def. 5.10.
- $p^S(u_1, \ldots, u_n) = \begin{cases} 1 & \text{if } \Theta \vdash_\Sigma p(u_1, \ldots, u_n) \\ 0 & \text{otherwise} \end{cases}$

Correspondingly, we can extend Def. 5.12:

Lemma 5.15. *In the situation of Def. 5.14 let Θ be complete and consistent. Then, we have*

$$[\![F]\!]^S = \begin{cases} 1 & \text{if } \Theta \vdash_\Sigma F \\ 0 & \text{if } \Theta \vdash_\Sigma \neg F \end{cases}$$

for all sentences $\vdash_\Sigma F : \texttt{form}$ that do not use $\forall$, $\exists$, or $\doteq$.

Proof. We use induction on F:

- Case $F = p(t_1, \ldots, t_n)$: We have

$$[\![p(t_1, \ldots, t_n)]\!]^S =^{Def} p^S([\![t_1]\!]^S, \ldots, [\![t_n]\!]^S) =^{\text{Lem. } 5.12} p^S(t_1, \ldots, t_n) =^{Def} \begin{cases} 1 & \text{if } \Theta \vdash_\Sigma p(u_1, \ldots, u_n) \\ 0 & \text{if } \Theta \vdash_\Sigma \neg p(u_1, \ldots, u_n) \end{cases}$$

  Here, the last step uses the fact that Θ is complete to turn "otherwise" into "if $\Theta \vdash_\Sigma \neg p(u_1, \ldots, u_n)$".

- Case $F = G \wedge H$: We distinguish two cases:
  - $[\![G]\!]^S = 1$ and $[\![H]\!]^S = 1$: We evaluate both sides of the equation:
    (left) We get $[\![G \wedge H]\!]^S = 1$.
    (right) The induction hypothesis for G and H yields $\Theta \vdash_\Sigma G$ and $\Theta \vdash_\Sigma H$. Thus, (using the conjunction introduction rule), we have $\Theta \vdash_\Sigma G \wedge H$. Thus, the right side also yields 1.
  - $[\![G]\!]^S = 0$ or $[\![H]\!]^S = 0$: We evaluate both sides of the equation:
    (left) We get $[\![G \wedge H]\!]^S = 0$.
    (right) The induction hypothesis for G or H yields $\Theta \vdash_\Sigma \neg G$ or $\Theta \vdash_\Sigma \neg H$. Thus, (using the conjunction elimination rules and negation introduction+elimination rules), we have $\Theta \vdash_\Sigma \neg(G \wedge H)$. Thus, the right side also yields 0.

- The cases for negation, disjunction, and implication proceed like the one for conjunction.

□

Remark 5.16. There is a more complex version of Lem. 5.15 that applies to all sentences. However, all of the above definitions have to be changed to use a different set $\mathtt{term}^S$. The key ideas is

- To handle equality, $\mathtt{term}^S$ is the quotient of the set of terms modulo the equivalence relation $\Theta \vdash_\Sigma t_1 \doteq t_2$. This is necessary to make sure $[\![t_1]\!]^S = [\![t_2]\!]^S = 1$ whenever $\Theta \vdash_\Sigma t_1 \doteq t_2$.
- To handle quantifiers, Σ and Θ have to be extended: If we can prove $\forall x_1 \ldots \forall x_n \; \exists x A$, we have to add an n-ary function symbols f and an axiom $\forall x_1 \ldots \forall x_n \; A[x/f(x_1, \ldots, x_n)$. This is necessary to make sure that $\mathtt{term}^S$ contains the element necessary for $[\![\exists x \; A]\!]$=1.

Finally, we can prove completeness of FOL:

Theorem 5.17 (Completeness of FOL)**.** *FOL is complete.*

Proof. We only prove the completeness of FOL with $\forall$, $\exists$, and $\doteq$ removed. The general prove proceeds similarly after generalizing Lem. 5.15.

By Lem. 5.5, we only have to show that every P-consistent FOL theory has a model. So assume a P-consistent theory $(\Sigma; \Theta)$.

Using Lem. 5.13, we can assume that Θ is complete. Otherwise, we construct a model of $\overline{\Theta}$, which will also be a model of Θ.

We use the term model from Def. 5.14. By Lem. 5.15, it satisfies all formulas in Θ. □

5.4 Incompleteness

In Ex. 3.54, we already saw that we cannot specify the natural numbers using Peano arithmetic in FOLEQ. We can now generalize this result.

Theorem 5.18. *The set SN^* from Ex. 3.54 is not recursively enumerable.*

Proof. Omitted. □

We immediately get the following corollary with far-reaching consequences:

Theorem 5.19 (Gödel's First Incompleteness Theorem)**.** *There is no logic in which there is a P-consistent theory whose*

- *signature includes the symbols of Peano arithmetic,*
- *whose theorems include SN^*,*
- *whose axioms are recursively enumerable.*

Proof. If there were such a logic and such a theory, we could recursively enumerate all its theorems and thus SN^*. □

This is one of the most famous results of computer science. It was discovered by Gödel in 1930 [?]. At around that time mathematicians were driven by Hilbert's program [?] to find a logic that can prove all theorems of mathematics. Gödel's incompleteness result showed that is not even possible to find a theory that can prove all theorems about the natural numbers (unless the theory is inconsistent, of course). Since many data types somehow contain the natural numbers, it implies a lot of negative results for other data types.

We can understand this as a special case of the philosophical map-territory problem: A map is different from the mapped territory. In some sense, the best map would have scale $1:1$ and be an exact copy of the territory; but that is impossible or at least impractical. Similarly, a theory is different from the intended model, and using an exact copy of the model would be impossible or impractical.

Gödel's result also imply a second incompleteness theorem: No matter what logic and what theory we use for mathematics, we will never be able to prove that the theory is consistent. In other words, no language can prove its own consistency. Consistency must be always be proved in a stronger language. But then we do not know whether that stronger language is consistent, and so on.

Therefore, it is theoretically possible that tomorrow someone will discover that set theory (see Ex. 1.32) is inconsistent. And then all of mathematics would break down. This is essentially what happened in 1897 and 1901, when Peano and Russell found inconsistencies in the theories used for mathematics. These theories had been mainly formalized by Frege in 1879 [?] and Cantor in 1883 [?] and had been used informally by all mathematicians before.

This gave rise to the development of modern set theories whose axioms are chosen very carefully to make inconsistency unlikely. The currently used variants of set theory have been developed mainly between 1900 and 1930, but mathematicians are still contributing important details. Parallel to the introduction of these new theories, researchers started studying logic and consistency in more detail. Then Gödel's result crushed all hope to settle the consistency question permanently.

Chapter 6

Computability

This chapter sketches some fundamental concepts from outside logic that are essential to make these notes self-contained.

Throughout this chapter, we fix an alphabet: a finite set Σ of symbols. Technically, it is perfectly sufficient to have a single symbol $\Sigma = \{0\}$. However, it is more convenient to use at least two symbols such as in $\Sigma = \{0, 1\}$ for theoretical analyses. In practice, we use much bigger alphabets such as ASCII with 2^8 symbols or Unicode with 2^{16} symbols (and more).

We look at the two fundamental concepts of computer science:

- the static part: sets of data,
- the dynamic part: functions between data.

Both concepts are inherently infinitary. Sets must be infinite to allow working with arbitrary data (e.g., the set of numbers or strings). And functions on infinite sets must be infinite themselves because they must map infinitely many inputs. But we can only ever actually compute with finite objects. Therefore, we need effective representations of infinite sets and functions.

6.1 Effective Representation of Functions

6.1.1 Computable Functions

Consider a function $f : \Sigma^* \to \Sigma^*$ mapping inputs i to outputs $o = f(i)$. We want to define f effectively, i.e., in a systematic way that allows other humans or machines to apply f to i and thus to compute o.

Therefore, we define:

Definition 6.1 (Algorithm)**.** An **algorithm** is a precisely described procedure, whose execution transforms words $i \in \Sigma^*$ to words $o \in \Sigma^*$ such that o only depends on i. (In particular o may not depend on when, where, and by whom the procedure is executed).

An algorithm A **terminates for an input** i, if the execution finishes after finitely many steps. In that case, we write $A(i)$ for the resulting output. An algorithm A **terminates**, if it terminates for all inputs.

Thus, every (terminating) algorithm induces a partial (total) function $A(-) : \Sigma^* \to \Sigma^*$.

Algorithms allow representing function systematically and reliably. That makes them the basis of all mechanic computation.

Terminology 6.2. Instead of algorithm, people may also say *(effective) method* or *(effective) procedure*. Here the word *effective* emphasizes that the descriptions must be systematically executable (by a human or a machine).

Remark 6.3 (Vagueness of the Definition of Algorithm). Def. 6.1 is not precise: It uses the words “precisely described procedure” and “execution”, which have not been defined previously. That is intentional. In retrospect, we can trace back the origin of computer science in the 1930s to this exact question: What is the description of a procedure, and what does it mean to execute it.

Nowadays, we have many different possible definitions. In particularly, every single programming language yields a rigorous definition: The programs are the descriptions, and running a program is its execution. We also call the former the syntax and the latter the semantics of the programming language.

Computer scientists have found many more possible definitions. These may use very diverse formalisms such as Turing machines, grammars, or automata. Even natural languages such as English or German are perfectly suitable for defining algorithms. (These descriptions are often called *pseudo-code*.)

Let us assume we have chosen one programming language or other formalism to make Def. 6.1 precise. The next question is whether we can give an algorithm for every function $\Sigma^* \to \Sigma^*$. We can immediately see that this is not the case: There are uncountably many such functions. But whatever formalism we use to define *algorithm* precisely, there can only be countably many algorithms. The reason is that the algorithms must be described in some language (even it is English), and languages can only have countably many words.

We call the functions that do have algorithms *computable*:

Definition 6.4 (Computable Function)**.** A partial function $f : \Sigma^* \to \Sigma^*$ is **computable** if there is an algorithm A such that A terminates for i iff $f(i)$ is defined and then $A(i) = f(i)$.

One might expect that different definitions of algorithm yield different computable functions. This is not the case:

Theorem 6.5 (Church-Turing Thesis)**.** *Every rigorous definitions of algorithm makes the same functions computable.*

Thus, the property of being computable is well-defined even though Def. 6.1 is vague.

Proof. Technically, we can never actually proof this because tomorrow someone might find a new definition of algorithm that is more powerful than everything we know.

But computer scientists have examined any number of possible definitions and have proved them all to be equivalent. □

Therefore, it does not matter which formalism we use to define algorithms. For example, any programming language is fine. In particular, whatever functions we can program in one programming language, we can also program in any other programming language.

6.1.2 Generalizations

It is easy to generalize computability to more general functions by choosing special alphabets.

Different Alphabets A functions $f : \Sigma^{\to}\Sigma'^*$ that uses multiple alphabets can be seen as a function f' that uses only the alphabet $\Sigma \cup \Sigma'$: $f'(i)$ is defined whenever $i \in \Sigma^*$ and then $f'(i) = f(i)$.

Therefore, we also speak of computable functions if the involved alphabets are different.

Multiple Arguments An n-ary function $f : \Sigma^* \times \ldots \Sigma^* \to \Sigma^*$ can be seen as a unary function f' on $\Sigma \cup \{|\}$: We define $f'(i_1| \ldots |i_n) = f(i_1, \ldots, i_n)$.

Therefore, we also speak of computable n-ary functions.

Concrete Sets Most mathematically relevant countable sets can be written as Σ^* (or appropriate subsets). For example, we can consider $\mathbb{N}$ to be the set $\{|\}^*$ or an appropriate subset of $\{0, 1\}^*$ or of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$. Therefore, we also speak of computable functions on the natural numbers.

Computable functions involving $\mathbb{Z}$ and $\mathbb{Q}$ are obtained similarly by adding symbols for $-$ and $/$.

We can never represent compute with all of $\mathbb{R}$ or $\mathbb{C}$ though because they are not countable.

6.1.3 Closure Properties

Most practically relevant functions are computable:

Theorem 6.6. *The following functions are computable:*

- *the identity function $x \mapsto x$*
- *for every $c \in \Sigma^*$, the constant function $x \mapsto c$*
- *for computable functions f, g, the composition $x \mapsto f(g(x))$*
- *for computable functions $f : \Sigma^* \to \{1, \ldots, n\}$ and $g_1, \ldots, g_n : \Sigma^* \to \Sigma^*$ the case-based function*

$$x \mapsto \begin{cases} g_1(x) & \text{if } f(x) = 1 \\ \vdots & \\ g_n(x) & \text{if } f(x) = n \end{cases}$$

- *for computable functions* $\mathrm{case} : \Sigma^* \to \{0,1\}$, $\mathrm{base} : \Sigma^* \to \Sigma^*$, $\mathrm{next} : \Sigma^* \to \Sigma^*$, *and* $\mathrm{step} : \Sigma^* \times \Sigma^* \to \Sigma^*$ *the recursive function r given by the following program*

$$\texttt{define}\ r(x) := \texttt{if}\ \mathrm{case}(x) = 0\ \texttt{then}\ \mathrm{base}(x)\ \texttt{else}\ \mathrm{step}(x, r(\mathrm{next}(x)))$$

Proof. It is easy to implement these functions in any programming language. □

Theorem 6.7. *All computable functions can be obtained by combining the operations from Thm. 6.6.*

Proof. We only sketch the proof.

First, we build a minimal programming language M out of the operations: identity corresponds to variables x, constants to values $c \in \Sigma^*$, composition to sequential operation ;, case-based function to case-split using an `if` or `switch` command, and recursion to recursive functions.

Second, we show that every programming language P can be reduced to M. We can do that by implementing an interpreter for P in M. (Or, more realistically, we proceed step-wise: We implement M_2 in M, and M_3 in M_2, and so on until we reach P.) □

6.2 Effective Representation of Sets

6.2.1 Decidable and Enumerable Sets

We can easily represent the set Σ^*: It is the set of words over Σ, i.e., the set of sequences of symbols. In a programming language, we usually call them strings. But we almost never need Σ^* — we usually need certain subsets of Σ^*. For example, the set of all well-formed $\mathcal{FOL}$-formulas is a subset of Σ^* for $\Sigma = \{\wedge, \vee, \Rightarrow, \neg, \forall, \exists\} \cup \{a, \ldots, z\} \cup \{(,)\} \cup \{,\}$. Similarly, every programming language is a subset of the set of all strings.

A set is defined by its elements. But there are two natural ways to represent effectively what the elements of a set are:

Definition 6.8 (Decidable Sets)**.** A set $S \subseteq \Sigma^*$ is called **decidable** if there is a computable total function f such that $f(i) = 1$ if $i \in S$ and $f(i) = 0$ otherwise. (Here, 0 and 1 are two arbitrary different elements of Σ^* serving as truth values.)

Definition 6.9 (Enumerable Sets)**.** A set $S \subseteq \Sigma^*$ is called **enumerable** if there is a computable total function f such that $\mathrm{image}(f) = S$.

If S is decidable, we have an effective way to test for being an element of S: We can execute $f(i)$ to determine whether $i \in S$ or not. We also call f a decision procedure for S.

If S is enumerable, we have an effective way to list all the elements of S: We can run f on all possible inputs and list all the outputs. We also call f an enumeration of S.

Terminology 6.10. Instead of enumerable, people may also say *recursively enumerable* or *semi-decidable*. (See Thm. 6.11 and 6.12 to understand why *semi-decidable* makes sense.)

Note that *denumerable* (also called countable) is different from enumerable.[1] A set S is denumerable if there is *some* function with domain $\mathbb{N}$ and image S; a set is enumerable if there is a *computable* function with domain $\mathbb{N}$ and image S. In particular, all subsets of Σ^* are denumerable.

6.2.2 Generalizations

Like in Sect. 6.1.2, we extend the concepts to all sets that can be represented using an appropriate alphabet. Thus, we speak of decidable and enumerable subsets of $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$.

It is easy to represent Cartesian products of sets by using an alphabet $\Sigma \cup \{|\}$. Therefore, we also speak of decidable and enumerable subsets of Cartesian products.

An important special case is the set of computable functions. Because the computable functions can be represented as programs in some programming language, we can see each computable function as an element of Σ^* where Σ is the alphabet of the programming language. Consequently, we can speak of decidable and enumerable sets of computable functions.

6.2.3 Closure Properties

Enumerable and decidable are not equivalent. Being able to decide is stronger because it lets us enumerate both the elements that are in S and those that are not in S:

Theorem 6.11. *A set S is decidable iff both S and $\Sigma^* \setminus S$ are enumerable.*

Proof. Left-to-right: Let f be a decision procedure for S. To enumerate S, we return i iff $f(i) = 1$ and nothing otherwise. To enumerate $\Sigma^* \setminus S$, we return i iff $f(i) = 0$ and nothing otherwise.

Right-to-left: Let f^+ and f^- be enumerations for S and $\Sigma^* \setminus S$, respectively. Let $x_1, x_2, \ldots$ be all the elements of Σ^*. To decide S, we take input i and compute $f^+(x_1), f^-(x_1), f^+(x_2), f^-(x_2), \ldots$ until we find i. We return 1 or 0 depending on whether we found i in f^+ or in f^-. □

If a set is enumerable, it is not necessarily decidable. Given an enumeration f^+, we might try to decide $i \in S$ by computing $f^+(x_1), f^+(x_2), \ldots$. If we find i in this sequence, we can return 1. But it would take infinitely long to determine that i is not in this sequence. Therefore, we can never return 0. The following theorem makes this more precise:

Theorem 6.12. *The following are equivalent:*

- *There is a total computable function f with whose image is S (i.e., S is enumerable).*
- *There is a partial computable function f with whose image is S.*
- *There is a partial computable function f such that $f(i) = 1$ iff $i \in S$ and $f(i)$ is undefined otherwise.*

Proof. It is straightforward to give the necessary enumerations. □

Theorem 6.13 (Closure Properties of Decidable Sets)**.** *The following sets are decidable:*

- *$\varnothing$ and Σ^**
- *all finite or cofinite[2] sets*
- *the intersection of decidable sets*
- *the union of decidable sets*
- *the complement of a decidable set*

Proof. It is straightforward to give the necessary decision procedures. □

Theorem 6.14 (Closure Properties of Enumerable Sets)**.** *The following sets are enumerable:*

- *$\varnothing$ and Σ^**

- *all finite or cofinite sets*
- *the union of enumerable sets*

Proof. It is straightforward to give the necessary enumerations. □

It is easy to understand why the enumerable sets are not closed under intersection and complement. If we tried to enumerate them, we would run into the problem that we have to wait infinitely long to conclude anything.

Note that neither decidable nor enumerable sets are closed under subsets. For example, a subset of a decidable set may be undecidable.

6.2.4 Counter-Examples

Some of the deepest results in computer science are about demonstrating that certain sets are not decidable or not enumerable.

The most important ones are about computation itself:

- Halting problem: Consider pairs (f, i) where f is a computable function and i an input for f. The set of all pairs (f, i) such that $f(i)$ is defined (i.e., such that computation of $f(i)$ terminates) is enumerable but not decidable.
- The set of computable functions f such that $f(i)$ is defined for all inputs (i.e., the set of terminating programs) is not enumerable.

In logic, we also encounter some critical examples:

- The set of theorems over a $\mathcal{FOLEQ}$-theory is enumerable but not necessarily decidable. For some special theories, decidability holds, but this is a very rare situation and usually non-trivial to prove.
- The set of consistent theories of $\mathcal{FOLEQ}$ is undecidable.
- The same applies to virtually any other logic with the exception of $\mathcal{PL}$. For $\mathcal{PL}$, consistency and theoremhood are decidable.
- The set of true sentences over the natural numbers is not enumerable. (See Thm. 5.18.)
- The same applies to most other interesting models.

Chapter 7

Closures and Galois Connections

7.1 Closures

7.2 Galois Connections

Chapter 8

Induction

The prototypical example of induction is mathematical induction, i.e., induction on the natural numbers. In computer science, induction is a more general principle about "doing something exactly once for all elements of a given set A". The "something" can be either of two things: a definition or a proof.

In a definition by induction, we define the function value $f(a)$ for all $a \in A$ by case distinction on a. Then we use the induction principle to argue that f is well-defined, i.e., that each $a \in A$ is covered by exactly one case.

In a proof by induction, we prove the property $P(a)$ for all $a \in A$ by case distinction on a. Then we use the induction principle to argue that P is proved, i.e., that each element of A is covered by exactly one case.

8.1 Mathematical Induction

The induction principle for the natural numbers $\mathbb{N}$ rests on the following:

Definition 8.1 (Natural Numbers). $\mathbb{N}$ is defined as follows:

1. $0 \in \mathbb{N}$
2. if $n \in \mathbb{N}$, then $s(n) \in \mathbb{N}$ (We call $s(n)$ the successor of n, e.g., $s(0) = 1$.)
3. The natural numbers are exactly the ones constructed by the above two cases, i.e.,
   (a) All objects obtained by the cases are different: $0 \neq s(n)$ for any n, and $s(n) \neq s(n')$ for any $n \neq n'$.
   (b) Every $n \in \mathbb{N}$ is obtained by one the cases: for every $n \in \mathbb{N}$, we have $n = 0$ or $n = s(n')$ for some $n' \in \mathbb{N}$.

Inductive Definition Consequently, we can define a function $f \in B^{\mathbb{N}}$ by giving $f(n) \in B$ by induction on $n \in \mathbb{N}$. We have to give two cases

- a case $f(0) \in B$,
- a case $f(s(n))$ where we may use the value $f(n) \in B$.

More formally, we have:

Theorem 8.2 (Definition by Induction). *Given a value $V_0 \in B$ and a function $V_s \in B^B$, the function $f \in B^{\mathbb{N}}$ given by*

$$f(0) = V_0$$

$$f(s(n)) = V_s(f(n))$$

is well-defined.

Example 8.3 (Addition). We define $m+n$ by induction on m.
Technically, this means we define a function f from $\mathbb{N}$ to $B=\mathbb{N}^{\mathbb{N}}$. f is the curried version of addition, i.e., we put $m+n=f(m)(n)$.
The cases for f are

$$\begin{aligned} f(0) &= n \mapsto n \\ f(s(m)) &= n \mapsto s(f(m)(n)) \end{aligned}$$

A more intuitive way to write these cases is:

$$f(0)(n)=0+n=n$$

$$f(s(m))(n)=s(m)+n=s(m+n)$$

Example 8.4 (Multiplication). We define $m \cdot n$ by induction on m.
Left as an exercise.

Inductive Proof Similarly, we can prove a property P by proving $P(n)$ by induction on $n \in \mathbb{N}$. We have to give two cases

- a proof of $P(0)$,
- a proof of $P(s(n))$ where we may assume that $P(n)$ holds.

More formally, we have:

Theorem 8.5 (Proof by Induction). *Assume $P(0)$ and "for all $n \in \mathbb{N}$, if $P(n)$ then $P(s(n))$". Then $P(n)$ for all $n \in \mathbb{N}$.*

Example 8.6 (Addition from the Right). We prove $m+0=m$ (AddZeroRight) by induction on m:

- Case $m=0$. We have to prove $0+0=0$, which holds by applying the definition of $+$.
- Case $m=s(m')$. We have to prove that if $m'+0=m'$ (IH), then $s(m')+0=s(m')$. We prove this by

  $$s(m')+0 =^{\text{def}} s(m'+0) =^{IH} s(m')$$

  where def refers to the definition of $+$.

Similarly, we can show that $m+s(n)=s(m+n)$ (AddSuccRight) by induction on m:

- Case $m=0$. We have to prove $0+s(n)=s(0+n)$, which holds by applying the definition of $+$ twice.
- Case $m=s(m')$. We have to prove that if $m'+s(n)=s(m'+n)$ (IH), then $s(m')+s(n)=s(s(m')+n)$. We prove this by

  $$s(m')+s(n) =^{\text{def}} s(m'+s(n)) =^{IH} s(s(m'+n))$$

  and

  $$s(s(m')+n) =^{\text{def}} s(s(m'+n))$$

Example 8.7 (Commutativity of Addition). We prove commutativity of addition $m+n=n+m$ by induction on m. Left as an exercise.

Example 8.8 (Associativity of Addition). We prove associativity of addition $l+(m+n)=(l+m)+n$ by induction on m.

- Case $m=0$. We have to prove $l+(0+n)=(l+0)+n$. We prove this by

  $$l+(0+n) =^{\text{def}} l+n$$

and

$$(l+0)+n =^{\text{AddSuccRight}} l+n$$

- Case $m = s(m')$. We have to prove that if $l+(m'+n) = (l+m')+n$ (IH) then $l+(s(m')+n) = (l+s(m'))+n$. We prove it as follows:

$$l+(s(m')+n) =^{\text{def}} l+s(m'+n) =^{\text{AddSuccRight}} s(l+(m'+n))$$

and

$$(l+s(m'))+n =^{\text{AddSuccRight}} s(l+m')+n =^{\text{def}} s((l+m')+n)$$

and these two are equal due to (IH).

Example 8.9 (Multiplication from the Right). We prove $m \cdot 0 = 0$ (MultZeroRight) by induction on m.
Similarly, we can show that $m \cdot s(n) = m \cdot n + m$ (MultSuccRight) by induction on m.
Left as an exercise.

Example 8.10 (Commutativity of Multiplication). We prove commutativity of addition $m \cdot n = n \cdot m$ by induction on m.
Left as an exercise.

Example 8.11 (Distributivity of Multiplication over Addition). We prove left-distributivity $l \cdot (m+n) = l \cdot m + l \cdot n$ by induction on l.
Left as an exercise.
Then we immediately have right-distributivity $(m+n) \cdot l = m \cdot l + n \cdot l$ using left-distributivity and commutativity.

Example 8.12 (Associativity of Multiplication). We prove associativity of multiplication $l \cdot (m \cdot n) = (l \cdot m) \cdot n$ by induction on m.
Left as an exercise.

Example 8.13 (Neutral Element of Multiplication). We put $1 = s(0)$ and prove $1 \cdot m = m$ and $m \cdot 1 = m$ (no induction necessary).
Left as an exercise.

8.2 Regular Induction

Given an alphabet Σ, we have the following induction principle for the set Σ^*:

Definition 8.14. Σ^* is defined by

- $\varepsilon \in \Sigma^*$,
- for every $x \in \Sigma$: if $w \in \Sigma^*$, then $xw \in \Sigma^*$,
- The elements of Σ^* are exactly the objects obtained by the above cases.

Note that the number of cases is $|\Sigma| + 1$.

Inductive Definition We give some examples for inductive definitions:

Example 8.15 (Reversal). We define the reversal w^R by induction on w

$$\varepsilon^R = \varepsilon$$

$$(xw)^R = w^R x$$

Example 8.16 (Length). We define the length $|w|$ by induction on w

$$|\varepsilon| = 0$$

$$|xw| = s(|w|)$$

Example 8.17 (DFA Transition Function). Given a DFA with transition function $\delta(q, x) \in Q$, we define $\delta^*(q, w)$ by induction on w:

$$\delta^*(q, \varepsilon) = q$$

$$\delta^*(q, xw) = \delta^*(\delta(q, x), w)$$

Inductive Proof We give some examples for inductive proofs:

Example 8.18 (Reversal with Symbols on the Right). We prove $(wy)^R = yw^R$ for $y \in \Sigma$ by induction on w.

- Case $w = \varepsilon$: We have to prove $(\varepsilon y)^R = y\varepsilon^R$. This holds because both sides are equal to y.
- Case $w = xw'$. We have to prove that if $(w'y)^R = yw'^R$, then $(xw'y)^R = y(xw')^R$. We prove it as follows:

  $$(xw'y)^R =^{\text{def}} (w'y)^R x =^{IH} yw'^R x$$

  and

  $$y(xw')^R =^{\text{def}} yw'^R x$$

Example 8.19 (Length with Symbols on the Right). We prove $|wx| = s(|w|)$ (LengthRight) by induction on w.
Left as an exercise.

Example 8.20 (Length-Preservation of Reversal). We prove $|w^R| = |w|$ by induction on w.
Left as an exercise.

Example 8.21 (Self-Inverseness of Reversal). We prove $(w^R)^R = w$ by induction on w.

- Case $w = \varepsilon$: We have to prove $(\varepsilon^R)^R = \varepsilon$, which we obtain by applying the definition of reversal twice.
- Case $w = xw'$. We have to prove that if $(w'^R)^R = w'$ (IH), then $((xw')^R)^R = xw'$. This is left as an exercise.

Regular Induction as a Special Case of Mathematical Induction Computer science prefers using induction on Σ^* as a stand-alone principle.

Mathematics prefers reducing it to induction on natural numbers. Then induction on the words $w \in \Sigma^*$ becomes induction on the length $l = |w|$ of w. The case $l = 0$ corresponds to the case $w = \varepsilon$. And the case $l = s(l')$ corresponds to the case $w = xw'$.

8.3 Context-Free Induction

Given an unambiguous context-free grammar (V, Σ, P, S), the context-free induction principle rests on the following:

Definition 8.22 (Produced Language)**.** For every $A \in V$, the set of words $L(A) \subseteq \Sigma^*$ is defined by

- For every production $A \to w_0 A_1 w_1 \ldots w_{n-1} A_n w_n \in P$ (where $A_i \in V$ and $w_i \in \Sigma^*$): if $v_i \in L(A_i)$, then $w_0 v_1 w_1 \ldots w_{n-1} v_n w_n \in L(A)$,
- The words in $L(A)$ are exactly the ones obtained by the above cases.

Remark 8.23. Note that the context-free induction principle defines finitely many sets at the same time: $L(A)$ for every $A \in V$. Typically we are only interested in $L(S)$ where S is the start symbol. But it is not possible to define only $L(S)$ — the definition of $L(S)$ must refer to $L(A)$ for other non-terminals A. Therefore, we define $L(A)$ for all $A \in V$ together. We speak of *mutual induction*.

Remark 8.24. If the grammar is ambiguous, then we cannot say "exactly" in Def. 8.22. The cases still cover all words, but some words are covered multiple times. This is harmless for proofs by induction (proving something twice is harmless) but illegal for definition by induction (defining something twice is only allowed if both definitions are the same).
Therefore, we require an unambiguous grammar.

Inductive Definition Given a function

$$f \in \mathcal{SET}^V$$

which maps every non-terminal symbol to a set, we define a set of functions $f_A : f(A)^{L(A)}$, i.e., one function for every non-terminal symbol A. Each function f_A maps the words $w \in L(A)$ produced from A to elements of the set $f(A)$.
The cases of a mutually inductive definition of the f_A are

- for every production $A \to w_0 A_1 w_1 \ldots w_{n-1} A_n w_n \;\in\; P$: a value $f_A(w_0 v_1 w_1 \ldots w_{n-1} v_n w_n) \in f(A)$ where we may use the values $f_{A_i}(v_i) \in f(A_i)$.

Example 8.25 (Semantics of Propositional Logic). Def. 3.2 is a simple example. Here `form` is the only non-terminal symbol, so we only define one function $[\![-]\!] = f_{\mathtt{form}}$ from $L(\mathtt{form})$ to $f(\mathtt{form}) = \{0,1\}$.

Example 8.26 (Free Variables). Def. 1.13 uses context-free induction to define FV. There are two non-terminals and both $FV(\mathtt{term}) = FV(\mathtt{form}) = \mathcal{P}(\mathrm{Var})$ where Var is the set of possible variable names. Then Def. 1.13 gives one case per production.

Example 8.27 (Serialization). The serialization of Bonus Exercise 1 is a typical example.
Here for every $A \in V$, $f(A)$ is the set of strings. The functions f_A map every word $w \in L(A)$ to its string representation.

Example 8.28 (Regular Induction). If we specialize to a right-linear grammar and require $f(A)$ to be the same set for all $A \in V$, we obtain regular induction as a special case.

Example 8.29 (Mathematical Induction). We obtain mathematical induction on the natural numbers as the special case of context-free induction on the grammar

$$N \quad ::= \quad 0 \quad | \quad succ(N)$$

Inductive Proofs Inductive proofs proceed very similarly. We do not give details here. However, keep in mind that all examples from Sect. 8.1 are examples of context-free induction via Ex. 8.29.

Context-Free Induction as a Special Case of Mathematical Induction Like regular induction, we can reduce context-free induction to induction on the natural numbers. The key idea is to use induction on the length of the derivation of a word. Alternatively, we can use induction on the height (= length of the longest branch) of the parse tree.

8.4 Context-Sensitive Induction

Context-sensitive induction is a generalization of context-free induction. Instead of functions $f_A \in f(A)^{L(A)}$, we use functions $f_A \in f(A)^{\mathrm{Cont} \times L(A)}$ where Cont is the context.

The context may be different for every inductive definition. Typically, it is a list of identifiers that have been declared together with additional information about every identifier. The context changes during the induction; in particular, declarations are added to the context when they are encountered.

Remark 8.30. Usually, we are only interested in the function $f_S(\varnothing, w)$ where S is the start symbol and $\varnothing$ the empty context. But just like context-free induction requires a function for every non-terminal, context-sensitive induction additionally requires taking an arbitrary context as an argument.

Inductive Definition Context-sensitive inductive definitions abound in theoretical computer science. Many non-trivial operations — e.g., compiling a program — are implemented in this way (even if that is not always obvious or mentioned explicitly).

Example 8.31 (Substitution). Def. 1.24 uses context-sensitive induction to define substitution application.

The induction context is a FOL-substitution γ, which stores which variables have been declared and which term each variable is substituted with.

$f(\mathtt{term})$ is the set of terms, and $f(\mathtt{form})$ is the set of formulas. The functions $f_{\mathtt{term}}(\gamma, t)$ and $f_{\mathtt{form}}(\gamma, F)$ are written $\overline{\gamma}(t)$ and $\overline{\gamma}(F)$.

Def. 1.24 gives one case per production. Note how the context is used in (and only in) the case $\overline{\gamma}(x)$.

Inference systems are the prototypical example of context-sensitive induction:

Example 8.32 (Well-Formed Syntax). Def. 2.5 uses context-sensitive induction to define the well-formed terms and formulas for a fixed signature Σ.

The induction context is a FOL-context Γ, which stores which variables have been declared.

The sets $f(\mathtt{term})$ and $f(\mathtt{form})$ contain boolean truth values, i.e., both $f_{\mathtt{term}}$ and $f_{\mathtt{form}}$ return either "holds" or "does not hold". We write $\Gamma \vdash_\Sigma t : \mathtt{term}$ or $\Gamma \vdash_\Sigma F : \mathtt{form}$ if $f_{\mathtt{term}}(\Gamma, t)$ holds or $f_{\mathtt{form}}(\Gamma, F)$ holds, respectively.

The inference rules of Fig. 2.1 give one case per production. Note how the context is used in (and only in) the case $\Gamma \vdash_\Sigma x : \mathtt{term}$.

Example 8.33 (Static Analysis). The static analysis from Bonus Exercise 1 uses context-sensitive induction. The context stores the declared identifiers and for every identifier its type.

The return type of the functions varies: For example, $f_{\mathtt{program}}(\Gamma, E)$ is a boolean; and $f_{\mathtt{expression}}(\Gamma, E)$ is a function that takes the expected type of E and returns a boolean.

Example 8.34 (Interpretation). The interpretation from Bonus Exercise 1 uses context-sensitive induction. The context stores the declared identifiers and for every identifier its type and its current value.

The return type of the functions varies: $f(\mathtt{program})$ is irrelevant/empty and $f(\mathtt{expression})$ is the set containing all string, integer, and boolean values of the underlying programming language.

8.5 Context-Free Induction in First-Order Logic

The term language of first-order logic is great for working with context-free grammars. In fact, we can think of FOLEQ as an abstract syntax framework for context-free grammars.

There is only one caveat: There may only be one non-terminal symbol. However, there is a straightforward generalization of FOLEQ (called typed FOLEQ or sorted FOLEQ) that can use a separate type for each non-terminal symbol.

But the restriction to a single non-terminal is not so bad: It already covers some of the most important context-free grammars.

Example 8.35 (Natural Numbers (Continuing Ex. 8.29)). The grammar

$$N \quad ::= \quad 0 \quad | \quad succ(N)$$

corresponds the FOLEQ signature from Ex. 1.35.

Example 8.36 (Words). The language of words over an alphabet $A = \{a_1, \dots, a_n\}$ can be written as the context-free grammar

$$W \quad ::= \quad \varepsilon \quad | \quad a_1 W \quad | \quad \dots \quad | \quad a_n W$$

It corresponds to the FOLEQ signature using

- nullary function symbol ε
- unary function symbols $\boldsymbol{a_1}$, …, $\boldsymbol{a_n}$, written as prefix without brackets, i.e., $\boldsymbol{a}w$ instead of $\boldsymbol{a}(w)$.

8.5.1 Simple Definitions in First-Order Logic

FOLEQ does not permit inductive definitions directly. In fact, FOLEQ signatures do not permit *any* definition at all. But we can give almost any definition indirectly using axioms in FOLEQ theories.

Constant Definitions For some term $\vdash_\Sigma t : \mathtt{term}$, the definition $c := t$ can be written in a FOLEQ theory by adding

- a nullary function symbol c,
- the axiom $c \doteq t$

Example 8.37 (Continuing Ex. 8.35). We define $1 := \mathit{succ}(0)$ by adding

- the nullary function symbol 1,
- the axiom $1 \doteq \mathit{succ}(0)$

Function Definitions For some term $x_1, \dots, x_n \vdash_\Sigma t : \mathtt{term}$, the definition f is the function that maps $x_1, \dots, x_n$ to t can be written in a FOLEQ theory by adding

- an n-ary function symbol f,
- the axiom $\forall x_1 \dots \forall x_n \ f(x_1, \dots, x_n) \doteq t$

Note that constant definitions can be seen of the special case of a function definition where $n = 0$.

Example 8.38 (Continuing Ex. 8.36). We define the doubling ww of a word using the term $w \vdash_\Sigma ww : \mathtt{term}$ by adding

- the unary function symbol *double*,
- the axiom $\forall w \ \mathit{double}(w) \doteq ww$

Predicate Definitions For some formula $x_1, \dots, x_n \vdash_\Sigma F : \mathtt{form}$, the definition p is the property that holds for $x_1, \dots, x_n$ if F holds can be written in a FOLEQ theory by adding

- an n-ary predicate symbol p,
- the axiom $\forall x_1 \dots \forall x_n \ p(x_1, \dots, x_n) \leftrightarrow t$

Example 8.39 (Continuing Ex. 8.35). We define the property *nonzero* of a natural number using the formula $n \vdash_\Sigma \neg n \doteq 0 : \mathtt{form}$ by adding

- the unary predicate symbol *nonzero*,
- the axiom $\forall n \ \mathit{nonzero}(n) \leftrightarrow \neg n \doteq 0$

8.5.2 Inductive Definitions in First-Order Logic

We obtain much more flexible options for definitions if we use induction. The basic idea is the same: We use axioms to capture the content of the definition. More precisely, we use one axiom for each case of the induction.

Inductive Function Definitions Functions are defined by adding a function symbol and then one axiom for every case.

Example 8.40 (Continuing Ex. 1.36, 8.3 and 8.35). We define the addition $m + n$ of natural numbers by adding

- the binary function symbol $+$ (written infix),
- the axioms $\forall n\ 0 + n \doteq n$ and $\forall m \forall n\ succ(m) + n \doteq succ(m + n)$.

And we define the multiplication $m \cdot n$ of natural numbers by adding

- the binary function symbol $\cdot$ (written infix),
- the axioms $\forall n\ 0 \cdot n \doteq 0$ and $\forall m \forall n\ succ(m) \cdot n \doteq (m \cdot n) + n$.

Example 8.41 (Continuing Ex. 8.15 and 8.36). We define the reversal of a word by adding

- the unary function symbol R (written as postfix superscript, i.e., w^R),
- the axioms $\forall w\ \boldsymbol{\varepsilon}^R \doteq \boldsymbol{\varepsilon}$ and $\forall w\ (\boldsymbol{a}w)^R \doteq w^R\boldsymbol{a}$ for all unary symbols $\boldsymbol{a}$ from Ex. 8.36.

Inductive Predicate Definitions Properties/predicates are defined by adding a predicate symbols and one axiom for every case.

Example 8.42 (Continuing 8.35). We define the less-or-equal predicate $\leq$ between natural numbers by adding

- the binary predicate symbol $\leq$ (written infix),
- the axioms

$$\forall n\ 0 \leq n \leftrightarrow true \qquad \text{and} \qquad \forall m \forall n\ succ(m) \leq n \leftrightarrow \exists x\ n \doteq succ(x) \wedge m \leq x$$

Remark 8.43. Of course, we can simply write $0 \leq n$ instead of $0 \leq n \leftrightarrow true$. We use the longer version here to emphasize the general shape of inductive predicate definitions.

Mutually Inductive Definitions We can also state mutually inductive definitions of functions or predicates. We simply add multiple function/predicate symbols at once and give one axiom per case.

Example 8.44 (Continuing 8.35). We define the predicates *even* and *odd* on natural numbers by adding

- the unary predicate symbols *even* and *odd*,
- the axioms

$$\begin{array}{ccc} even(0) \leftrightarrow true & \text{and} & \forall n\ even(succ(n)) \leftrightarrow odd(n) \\ odd(0) \leftrightarrow false & \text{and} & \forall n\ odd(succ(n)) \leftrightarrow even(n) \end{array}$$

8.5.3 Inductive Proofs in First-Order Logic

We can now finally understand the motivation behind the Peano axioms from Ex. 1.35:

Example 8.45 (Induction Schemata in First-Order Logic). The Peano axioms from Ex. 1.35 arise as follows:

- The function symbols 0 and *succ* formalizes the cases 1 and 2 of Def. 8.1.
- The axioms for injectivity of *succ* and starting point 0 formalize the cases 3a of Def. 8.1. Respectively, they capture that no two successors are equal and that no successor is equal to 0.
- The axiom schema of induction approximately formalizes case 3b of Def. 8.1. It expresses the intuition that 0 and all successors make up the whole natural numbers.

Thus, the Peano axioms systematically formalize Def. 8.1 in FOLEQ.

Along these lines, we can define induction schemata in FOLEQ for all context-free grammars with a single non-terminal. Then we can revisit the examples of inductive proofs from Sect. 8.1, 8.2, and 8.3 and formally carry them out in FOLEQ.

Remark 8.46 (Incompleteness of Induction). Note that Ex. 8.45 says "approximately formalizes". The problem is that the axiom schema of induction yields only countably many axioms (because there are only countably many formulas). But there are uncountably many things we can do with the natural numbers (e.g., uncountably many different functions out of $\mathbb{N}$). Therefore, FOLEQ cannot fully capture case 3b of Def. 8.1.

Indeed, as we know from Ex. 3.54, even though the theory of Ex. 8.45 is complete (i.e., can prove or disprove every formula), there are non-standard models that satisfy the exact same FOLEQ formulas as the intended standard model SN.

Moreover, and even worse, we know that Peano arithmetic — i.e., the union of the theories from Ex. 8.40 and 8.45 — is not even a complete theory: There are formulas that are true in the standard model but can be neither proved nor disproved in FOLEQ. The proofs of these formulas require induction hypotheses that FOLEQ's Peano arithmetic cannot express.

Chapter 9

Morphisms

Generally, the term **morphism** (or homomorphism) refers to a structure-preserving translation between formal languages or objects. Typically, there is a collection of objects with a certain structure, and then for each two objects A, B there is a set of morphisms from A to B. Morphism are a crucial tool to study relations between objects: Suitable morphisms can be used to represent translations, inclusions, encodings, or isomorphisms. Furthermore, often only the definition of morphism makes apparent what the common structure of the objects is – namely that which is preserved by the morphisms.

In this section, we define morphism between signatures and theories. These extend to translations between expressions, models, and proofs.

Signature morphisms and substitutions are very similar. Their relation is summarized in the following table:

Symbol	Declared in	Translated by
logical symbol	logic L	(logic translation)
function/predicate symbol	L-signature Σ	morphism $\vdash^L \sigma : \Sigma \to \Sigma'$
variable	Σ-context Γ	substitution $\vdash^L_\Sigma \gamma : \Gamma \to \Gamma'$

9.1 Translations between Signatures

[1]

Definition 9.1 (FOL-Signature Morphisms)**.** Let $\Sigma = (\Sigma_f, \Sigma_p, ar)$ and $\Sigma' = (\Sigma'_f, \Sigma'_p, ar')$ be two FOL-signatures. A **signature morphism** from Σ to Σ' is a function σ mapping with domain $\Sigma_f \cup \Sigma_p$ such that

- for every function symbol $f \in \Sigma_f$ with $ar(f) = n$, we have

$$x_1, \ldots, x_n \vdash_{\Sigma'} \sigma(f) : \texttt{term}$$

- for every predicate symbol $p \in \Sigma_p$ with $ar(p) = n$, we have

$$x_1, \ldots, x_n \vdash_{\Sigma'} \sigma(p) : \texttt{form}$$

In that case, we write $\sigma : \Sigma \to \Sigma'$.

Intuitively, every n-ary Σ-function symbol is mapped to an n-ary Σ'-function, and every n-ary Σ-predicate symbol is mapped to an n-ary Σ'-formula. For readers familiar with Sect. 22.2.1, we can state this more precisely: A signature morphism maps each n-ary Σ-symbol to a Σ'-expression with n-holes.

[1]Remark for the 2011 lecture: These notes use $x_1, \ldots, x_n \vdash_\Sigma t : \texttt{term}$ to say that t is a well-formed term over the signature Σ whose free variables are among $x_1, \ldots, x_n$. Similarly, they use $x_1, \ldots, x_n \vdash_\Sigma F : \texttt{form}$ to say that F is a well-formed term over the signature Σ whose free variables are among $x_1, \ldots, x_n$. These are defined in Def. 2.5. They have the same meaning as the notations using wff_ι and wff_o.

Example 9.2. Consider the signatures of Σ of monoids with $\circ$ and e and Σ' of groups with $\circ$, e, and x^{-1}. A morphism from Σ to Σ' maps $\sigma(\circ) = \circ$ and $\sigma(e) = e$.

Example 9.3. We can define a signature morphism from the signature of monoids to itself that flips the operation: It maps e to itself (as a term with 0 free variables) and *comp* to $comp(x_2, x_1)$.

Remark 9.4. Often a specialized, simpler definition than Def. 9.1 is found in the literature. Then σ maps n-ary function/predicate symbols to n-ary functions/predicate *symbols*, i.e., it maps $\Sigma_f \to \Sigma'_f$ and $\Sigma_p \to \Sigma'_p$. If this definition is used, signature morphisms in the sense of Def. 9.1 are called generalized or derived signature morphisms.

9.2 Translating Syntax

Just like every signature Σ gives us an inductive definition of the set of well-formed Σ-expressions (i.e., terms and formulas), every signature morphism $\sigma : \Sigma \to \Sigma'$ gives us an inductive function that translates Σ-expressions to Σ'-expressions. This is a typical property of morphisms.

Definition 9.5 (Expression Translation)**.** Assume $\sigma : \Sigma \to \Sigma'$. σ induces a map $\overline{\sigma}$ from Σ-terms to Σ'-terms and from Σ-formulas to Σ'-formulas as follows:

- terms:
  - $\overline{\sigma}(x) = x$,
  - $\overline{\sigma}(f(t_1, \dots, t_n)) = \sigma(f)[x_1/\overline{\sigma}(t_1), \dots, x_n/\overline{\sigma}(t_n)]$,
- formulas:
  - $\overline{\sigma}(true) = true$,
  - $\overline{\sigma}(false) = false$,
  - $\overline{\sigma}(p(t_1, \dots, t_n)) = \sigma(p)[x_1/\overline{\sigma}(t_1), \dots, x_n/\overline{\sigma}(t_n)]$,
  - $\overline{\sigma}(t_1 \doteq t_2) = \overline{\sigma}(t_1) \doteq \overline{\sigma}(t_2)$,
  - $\overline{\sigma}(F \wedge G) = \overline{\sigma}(F) \wedge \overline{\sigma}(G)$ and similarly for the other connectives,
  - $\overline{\sigma}(\forall x\ F) = \forall x\ \overline{\sigma}(F)$ and similarly for the other quantifier.

Remark 9.6. $\overline{\sigma}$ is called the *homomorphic extension* of σ.

The structure that is preserved by signature morphisms is that of well-formed expressions, i.e., the syntax. More formally, we have the following lemma

Lemma 9.7 (Morphism Property)**.** *Assume* $\sigma : \Sigma \to \Sigma'$. *Then*

- *if* $\Gamma \vdash_\Sigma t : \texttt{term}$, *then* $\Gamma \vdash_{\Sigma'} \overline{\sigma}(t) : \texttt{term}$,
- *if* $\Gamma \vdash_\Sigma F : \texttt{form}$, *then* $\Gamma \vdash_{\Sigma'} \overline{\sigma}(F) : \texttt{form}$,

Proof. By an easy induction on t or F, respectively. □

Notation 9.8. We often write σ instead of $\overline{\sigma}$.

Definition 9.9 (FOL-Sentence Translation)**.** Just like $\mathbf{Sen}(\Sigma)$ is the set of sentences for every $\Sigma \in \mathbf{Sig}^{\mathcal{FOLEQ}}$, we can specialize Def. 9.5 to obtain a sentence translation function $\mathbf{Sen}^{\mathcal{FOLEQ}}(\sigma) : \mathbf{Sen}^{\mathcal{FOLEQ}}(\Sigma) \to \mathbf{Sen}^{\mathcal{FOLEQ}}(\Sigma')$ for every signature morphism $\sigma : \Sigma \to \Sigma'$:

$$\mathbf{Sen}^{\mathcal{FOLEQ}}(\sigma)(F) = \overline{\sigma}(F).$$

When dealing with sentence translations, the following notation is often useful to translate sets of sentences:

Notation 9.10. For $\Theta \subseteq \mathbf{Sen}(\Sigma)$ and a function $\varphi : \mathbf{Sen}(\Sigma) \to B$, we write $f(\Theta)$ for $\{\varphi(F) : F \in \Theta\}$.

9.3 Translating Proof-Theory

Signature morphisms also yield inductive translations of judgments and proofs.

Definition 9.11 (Proof Translation)**.** Assume $\sigma : \Sigma \to \Sigma'$. We define a map $\overline{\sigma}$ from Σ-judgments and Σ-proofs to Σ'-judgments and Σ'-proofs:

- Every hypothesis of a proof is mapped to a hypothesis by translating every expression occurring in it along $\overline{\sigma}$.
- Every application of a FOLEQ proof rule r to proofs $P_1, \ldots, P_n$ is mapped to the proof obtained by applying r to $\overline{\sigma}(P_1), \ldots, \overline{\sigma}(P_n)$. If r takes – term or formula – parameters, these are translated along $\overline{\sigma}(-)$ as well.

We leave working out the details as an exercise.

Remark 9.12. Just like for sentences, $\overline{\sigma}$ is a *homomorphic extension* of σ, and we often write σ instead of $\overline{\sigma}$.

The structure preserved by the proof theory translation is the proof-theoretical semantics: well-formed proofs are mapped to well-formed proofs:

Lemma 9.13 (Morphism Property)**.** *Assume $\sigma : \Sigma \to \Sigma'$. Then:*

- *if p is a proof of $\Theta \vdash_\Sigma F$, then $\overline{\sigma}(p)$ is a proof of $\overline{\sigma}(\Theta') \vdash_{\Sigma'} \overline{\sigma}(F)$,*
- *in particular: $\Theta \vdash_\Sigma F$ implies $\overline{\sigma}(\Theta') \vdash_{\Sigma'} \overline{\sigma}(F)$.*

Proof. By induction p and Γ, we prove that if p is a proof of $\Theta; \Gamma \vdash_\Sigma F$, then $\overline{\sigma}(p)$ is a proof of $\overline{\sigma}(\Theta'); \Gamma \vdash_{\Sigma'} \overline{\sigma}(F)$. The details are left as an exercise. □

Remark 9.14. Note the similarities between translating syntax and translating proof theory, specifically

- Def. 9.5 and 9.11,
- Def. 9.7 and 9.13.

9.4 Translating Model Theory

[2]

Just like every signature yields a collection of models, every signature morphism σ yields a map $-|_\sigma$ of Σ'-models to Σ-models:

Definition 9.15 (Model Translation)**.** Assume $\sigma : \Sigma \to \Sigma'$. Consider a model $I \in \mathbf{Mod}(\Sigma')$; we define the model $I|_\sigma \in \mathbf{Mod}(\Sigma)$ as follows:

- $i^{I|_\sigma} = i^I$,
- $f^{I|_\sigma}(u_1, \ldots, u_n) = [\![\sigma(f)]\!]^{I,[x_1/u_1,\ldots,x_n/u_n]}$ for $f \in \Sigma_f$ with $ar(f) = n$,
- $p^{I|_\sigma}(u_1, \ldots, u_n) = [\![\sigma(p)]\!]^{I,[x_1/u_1,\ldots,x_n/u_n]}$ for $p \in \Sigma_p$ with $ar(p) = n$.

[2] Remark for the 2011 lecture: These notes use the letter I for models and write $\texttt{term}^I$ for the universe of the model I. See Def. 3.9.

Remark 9.16. Def. 9.15 becomes a lot simpler and more intuitive if we consider the special case mentioned in Rem. 9.4. Then we have $i^{I|_\sigma} = \mathtt{term}^I$ and $f^{I|_\sigma} = \sigma(f)^I$ and $p^{I|_\sigma} = \sigma(p)^I$. In particular, recall that models like I and $I|_\sigma$ are just functions that map symbols to interpretation; then $I|_\sigma$ is just the restriction of I to a smaller domain.

Just like the translation of expression preserves the well-formedness, the translation of models preserves the semantics of expressions:

Lemma 9.17. *Assume $\sigma : \Sigma \to \Sigma'$, and a model $I \in \mathbf{Mod}(\Sigma')$. Then:*

- *for terms $\Gamma \vdash_\Sigma t : \mathtt{term}$ and an assignment α:*
  $$[\![t]\!]^{I|_\sigma,\alpha} = [\![\overline{\sigma}(t)]\!]^{I,\alpha}$$
  in particular, if t is closed:
  $$[\![t]\!]^{I|_\sigma} = [\![\overline{\sigma}(t)]\!]^{I}$$
- *for formulas $\Gamma \vdash_\Sigma F : \mathtt{form}$ and an assignment α:*
  $$[\![F]\!]^{I|_\sigma,\alpha} = [\![\overline{\sigma}(F)]\!]^{I,\alpha}$$
  in particular, if F is closed:
  $$[\![F]\!]^{I|_\sigma} = [\![\overline{\sigma}(F)]\!]^{I}$$
  $$I|_\sigma \models_\Sigma F \quad \text{iff} \quad I \models_{\Sigma'} \overline{\sigma}(F)$$

Proof. By induction on Γ and t or F, respectively. □

Remark 9.18. Note the similarities between translating syntax and translating model theory, specifically

- Def. 9.5 and 9.15 (except that the orientation is flipped),
- Def. 9.7 and 9.17.

9.5 An Abstract Definition of a Logic, revisited

Recall Def. 1.29, Def. 4.8, and 3.20 of logic syntax, model theory, and proof theory, respectively. We can now refine these definitions by adding morphisms everywhere.

In fact, our definitions are still not the final definitions because we will omit some advanced aspects. But the advanced aspects actually make the definitions more symmetric; therefore, we given them anyway but distinguish them from the rest.

Definition 9.19 (Logic Syntax)**.** A **logic syntax** is a pair $(\mathbf{Sig}, \mathbf{Sen})$ where

- **Sig** is
  - a collection of objects
    $\Sigma \in \mathbf{Sig}$ is called a *signature*
  - a family of sets such that $\mathbf{Sig}(\Sigma, \Sigma')$ is a set for all $\Sigma, \Sigma' \in \mathbf{Sig}$
    $\sigma \in \mathbf{Sig}(\Sigma, \Sigma')$ is called a *signature morphism* from Σ to Σ, which we also write as $\sigma : \Sigma \to \Sigma'$
- **Sen** is
  - a mapping from $\mathbf{Sig} \to \mathcal{SET}$, i.e., if $\Sigma \in \mathbf{Sig}$, then $\mathbf{Sen}(\Sigma)$ is a set
    $F \in \mathbf{Sen}(\Sigma)$ is called a Σ-*sentence*
  - a family of mappings $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ for every $\sigma : \Sigma \to \Sigma'$
    $\mathbf{Sen}(\sigma)(F)$ is called the *sentence translation* of F along σ

Remark 9.20. Note that a logic syntax has 2x2 components:

signatures Σ	signature morphisms $\sigma : \Sigma \to \Sigma'$
sentences $F \in \mathbf{Sen}(\Sigma)$	sentence translation $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$

Example 9.21. FOLEQ from Ex. 1.30 can be refined to an example of the refined definition of logic syntax:

- $\mathbf{Sig}^{\mathcal{FOLEQ}}$ is the collection of FOLEQ signatures as defined in Def. 1.4.
- $\mathbf{Sig}^{\mathcal{FOLEQ}}(\Sigma, \Sigma)$ is the set of FOLEQ signature morphisms as defined in Def. 9.1.
- $\mathbf{Sen}^{\mathcal{FOLEQ}}(\Sigma)$ is the set of FOLEQ sentences over Σ as defined in Def. 1.17.
- $\mathbf{Sen}^{\mathcal{FOLEQ}}(\sigma)$ is the sentence translation as defined in Def. 9.9.

Definition 9.22 (Proof Theory)**.** A proof theory for the logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ is a pair $(\mathbf{Pf}, \vdash)$ such that

- $\mathbf{Pf}$ is
  - a mapping such that for $\Sigma \in \mathbf{Sig}$
    - $\mathbf{Pf}(\Sigma)$ is a collection of objects

      $J \in \mathbf{Pf}\,\Sigma$ is called a Σ-*judgment*
    - a family of sets such that $\mathbf{Pf}(\Sigma)((J_1, \dots, J_m), J)$ is a set for all judgments of $J_1, \dots, J_m, J \in \mathbf{Pf}(\Sigma)$

      $p \in \mathbf{Pf}(\Sigma)((J_1, \dots, J_m), J)$ is called a Σ-*proof* of J using hypotheses $J_1, \dots, J_k$
  - a family of mapping such that for $\sigma : \Sigma \to \Sigma'$
    - $\mathbf{Pf}(\sigma) : \mathbf{Pf}(\Sigma) \to \mathbf{Pf}(\Sigma')$ is a mapping

      $\mathbf{Pf}(\sigma)(J)$ is called the *judgment translation* of J along σ
    - $\mathbf{Pf}(\sigma) : \mathbf{Pf}(\Sigma)((J_1, \dots, J_m), J) \to \mathbf{Pf}(\Sigma)((\mathbf{Pf}(\sigma)(J_1), \dots, \mathbf{Pf}(\sigma)(J_m)), \mathbf{Pf}(\sigma)(J))$

      $\mathbf{Pf}(\sigma)(p)$ is called the *proof translation* of p along σ
- $\vdash$ is
  - a family of mappings $\vdash_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Pf}(\Sigma)$ for every $\Sigma \in \mathbf{Sig}$

    $\vdash_\Sigma F$ is called the *validity judgment* for F
  - such that $\mathbf{Pf}(\sigma)(\vdash_\Sigma F) = \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ for $\sigma : \Sigma \to \Sigma'$

    $\vdash$ commutes with $\mathbf{Sen}(\sigma)/\mathbf{Pf}(\sigma)$

Remark 9.23. Note that a proof theory has 2x2 components, 3 definitions and 1 property:

judgments and proofs $\mathbf{Pf}(\Sigma)$	judgment and proof translation $\mathbf{Pf}(\sigma) : \mathbf{Pf}(\Sigma) \to \mathbf{Pf}(\Sigma')$
validity judgment $\vdash_\Sigma$: $\mathbf{Sen}(\Sigma) \to \mathbf{Pf}(\Sigma)$	commutation property for $\vdash$ wrt. **Sen** and **Pf**

Example 9.24. FOLEQ from Ex. 4.9 can be refined to an example of the refined definition of proof theory:

- $\mathbf{Pf}^{\mathcal{FOLEQ}}(\Sigma)$ consists of the FOLEQ judgments and proofs as defined in Def. 4.5.
- $\mathbf{Pf}^{\mathcal{FOLEQ}}(\sigma)$ translates proofs structurally as defined in Def. 9.11.
- $\vdash_\Sigma F$ is defined as $\varnothing; \varnothing \vdash_\Sigma F$.
- We omit showing the commutation property.

Definition 9.25 (Model Theory)**.** A model theory for the logic syntax $(\mathbf{Sig}, \mathbf{Sen})$ is a pair $(\mathbf{Mod}, \models)$ such that

- $\mathbf{Mod}$ is
  - a mapping such that for $\Sigma \in \mathbf{Sig}$
    - $\mathbf{Mod}(\Sigma)$ is a collection of objects

      $M \in \mathbf{Mod}\Sigma$ is called a Σ-*model*
  - a family of mapping such that for $\sigma : \Sigma \to \Sigma'$
    - $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ is a mapping

      $\mathbf{Mod}(\sigma)(M)$ is called the *reduction* of M along σ
- $\models$ is
  - a family of relations $\models_\Sigma \subseteq \mathbf{Sen}(\Sigma) \times \mathbf{Mod}(\Sigma)$ for every $\Sigma \in \mathbf{Sig}$ and $F \in \mathbf{Sen}(\Sigma)$

    $\models_\Sigma$ is called the *satisfaction relation*
  - such that $\mathbf{Mod}(\sigma)(M) \models_\Sigma F$ iff $M \models_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ for $\sigma : \Sigma \to \Sigma'$, $F \in \mathbf{Sen}(\Sigma)$, $M \in \mathbf{Mod}(\Sigma')$

    $\models$ commutes with $\mathbf{Sen}(\sigma)/\mathbf{Mod}(\sigma)$

Remark 9.26. Note that a model theory has 2x2 components, 3 definitions and 1 property:

models and model morphisms $\mathbf{Mod}(\Sigma)$	model (morphism) reduction $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$
satisfaction relation $\models_\Sigma \subseteq \mathbf{Sen}(\Sigma) \times \mathbf{Mod}(\Sigma)$	commutation property for $\models$ wrt. $\mathbf{Sen}$ and $\mathbf{Mod}$

Example 9.27. FOLEQ from Ex. 3.22 can be refined to an example of the refined definition of model theory:

- $\mathbf{Mod}^{\mathcal{FOLEQ}}(\Sigma)$ consists of the FOLEQ models as defined in Def. 3.9 and the FOLEQ model morphisms as indicated in Sect. 10.1.
- $\mathbf{Mod}^{\mathcal{FOLEQ}}(\sigma)(I) = I|_\sigma$ is the model reduction as defined in Def. 9.15.
- $\models_\Sigma$ is defined as in Def. 3.12.
- We omit showing the commutation property.

Finally, we can extend Def. 5.1 appropriately:

Definition 9.28 (Logic)**.** A *logic* consists of

- a logic syntax $(\mathbf{Sig}, \mathbf{Sen})$,
- a model theory $(\mathbf{Mod}, \models)$ for $(\mathbf{Sig}, \mathbf{Sen})$,
- a proof theory $(\mathbf{Pf}, \vdash)$ for $(\mathbf{Sig}, \mathbf{Sen})$.

Example 9.29. FOLEQ is the logic $(\mathbf{Sig}^{\mathcal{FOLEQ}}, \mathbf{Sen}^{\mathcal{FOLEQ}}, \mathbf{Mod}^{\mathcal{FOLEQ}}, \models^{\mathcal{FOLEQ}}, \mathbf{Pf}^{\mathcal{FOLEQ}}, \vdash^{\mathcal{FOLEQ}})$.

9.6 Theory Morphisms

We can extend the notion of signature morphisms to theory morphisms. Just like a signature morphism translates syntax, proof theory, and model theory between signatures, a theory morphism translates them between theories.

Definition 9.30 (Theory Morphisms)**.** $\sigma : \Sigma \to \Sigma'$ is called a proof theoretical **theory morphism** from (Σ, Θ) to (Σ', Θ') if $\overline{\sigma}$ maps every $F \in \Theta$ to a proof theoretical theorem $\Theta' \vdash_{\Sigma'} \sigma(F)$.
$\sigma : \Sigma \to \Sigma'$ is called a model theoretical **theory morphism** from (Σ, Θ) to (Σ', Θ') if $\overline{\sigma}$ maps every $F \in \Theta$ to a model theoretical theorem $\Theta' \models_{\Sigma'} \sigma(F)$.

Notation 9.31. We usually do not distinguish proof and model theoretical theory morphisms and write both of them as $\sigma : (\Sigma, \Theta) \to (\Sigma', \Theta')$. If a distinction is necessary, we speak of P-theory morphisms and M-theory morphisms. See also Not. 3.37 and 4.14.

Remark 9.32. We know, of course, that in FOLEQ P- and M-theory morphisms are the same thing because FOLEQ is sound and complete. The definition is split into two parts here in order to apply to to arbitrary logics (including those where soundness or completeness do not hold).

Example 9.33. The signature morphism from monoids to groups is a FOLEQ-theory morphism. This is rather trivial because each $\sigma(F)$ is already an axiom of the theory of groups.

Example 9.34. The signature morphism from Ex. 9.3 is a FOLEQ-theory morphism. (Exercise!)

Example 9.35. For a larger example of lots of FOLEQ theory morphisms (all represented in Twelf, see Sect. 26), see [**?**].

The basic intuition behind theory morphisms is that they are signature morphisms that preserve the semantics. Therefore, we can show additional properties of the proof theory and the model theory translation:

Lemma 9.36 (Proof Translation)**.** *Assume a P-theory morphism $\sigma : (\Sigma, \Theta) \to (\Sigma', \Theta')$. Then:*

- *if p is a proof of $\Theta \vdash_\Sigma F$, then there is also a proof of $\Theta' \vdash_{\Sigma'} \overline{\sigma}(F)$,*
- *in particular: $\Theta \vdash_\Sigma F$ implies $\Theta' \vdash_{\Sigma'} \overline{\sigma}(F)$.*

Proof. We know $\overline{\sigma}(p)$ is a proof of $\overline{\sigma}(\Theta) \vdash_{\Sigma'} \overline{\sigma}(F)$. Each reference to an axiom $A \in \overline{\sigma}(F)$ can be replaced with a proof of $\Theta' \vdash_{\Sigma'} A$, which exists because σ is a P-theory morphisms. □

Lemma 9.37 (Model Translation)**.** *Assume an M-theory morphism $\sigma : (\Sigma, \Theta) \to (\Sigma', \Theta')$. Then:*

- *for every model $I \in \mathbf{Mod}(\Sigma')$: if $I \in \mathbf{Mod}(\Sigma'; \Theta')$, then $I|_\sigma \in \mathbf{Mod}(\Sigma; \Theta)$.*
- *in particular: $\Theta \models_\Sigma F$ implies $\Theta' \models_{\Sigma'} \overline{\sigma}(F)$.*

Proof. □

We can summarize and extend the above lemmas in the following theorem:

Theorem 9.38 (Theory Morphisms)**.** *Consider a logic L, two theories (Σ, Θ) and (Σ', Θ'), and $\sigma : \Sigma \to \Sigma'$. Then:*

1. *The following are equivalent:*
   (a) *σ is a P-theory morphism.*
   (b) *$\Theta' \vdash_{\Sigma'} \overline{\sigma}(F)$ for every $F \in \Theta$.*
   (c) *$\Theta' \vdash_{\Sigma'} \overline{\sigma}(F)$ for every $F \in Thm(\Sigma; \Theta)$.*
2. *The following are equivalent:*
   (a) *σ is an M-theory morphism.*
   (b) *$\Theta' \models_{\Sigma'} \overline{\sigma}(F)$ for every $F \in \Theta$.*

   (c) *$\Theta' \models_{\Sigma'} \overline{\sigma}(F)$ for every $F \in Thm(\Sigma;\Theta)$.*

   (d) *For every model $I \in \mathbf{Mod}(\Sigma')$: if $I \in \mathbf{Mod}(\Sigma';\Theta')$, then $I|_\sigma \in \mathbf{Mod}(\Sigma;\Theta)$.*

3. *if L is sound and σ is a P-theory morphism, then σ is an M-theory morphism,*

4. *if L is complete and σ is an M-theory morphism, then σ is an P-theory morphism.*

Proof. For the first equivalence, (1a) is equivalent to (1b) by definition; (1b) implies (1c) by Lem. 9.36; (1c) trivially implies (1b).

Similarly, for the second equivalence, (2a) is equivalent to (2b) by definition. (2b) implies (2c) by Lem. 9.36; (2c) trivially implies (2b). (2a) implies (2d) by Lem. 9.36; we omit the proof of (2d) implies (2a)

Finally, using soundness, (1b) implies (2b). And, using completeness, (2b) implies (1b). □

The importance of theory morphisms is that they can be used to transport theorems: Theorems proved in Σ can be moved to theorems in Σ' along a theory morphism. That can greatly increase the set of available theorems, especially when theory morphisms connect rather different fields of mathematics. The equivalence between (1b) and (1c) is important: We want all theorems to be mapped to theorems, but we only have to check that all axioms are mapped to theorems. That is crucial because the former is infinite, the latter usually finite.

Case (2c) is the most important to use: It tells us that all (Σ,Θ)-theorems can be moved to all (Σ',Θ')-models. Case (1b) is usually the easiest to establish.

Moreover, theory morphisms can be used to piece together theories from building blocks thus providing the basis of modular (and thus large-scale) development of theories. This latter is crucial when approaching logic from the perspective of computer science, and theory morphisms constitute the central technology to move towards something that could be called *theory engineering* akin to software engineering. Later we will see that logics can be represented as signatures of a suitable meta-logic, which we call a logical framework. Similarly, logic translations can be represented as signature morphisms. Thus, the above remark on theory engineering can be extended to *logic engineering*.

Chapter 10

Universal Model Theory

[1] In this chapter we investigate general methods for studying the collection of models of a given theory. The word "universal" refers to the insight that we can establish many deep definitions and theorems generically for an arbitrary theory.

This section will focus on $\mathcal{FOLEQ}$. But many results are even more general – they can be established for an arbitrary theory of an arbitrary logic. Unless mentioned otherwise, we work with a fixed $\mathcal{FOLEQ}$-signature Σ.

The basic idea that Σ-models are sets with structure (namely the structure given by the interpretations of the Σ-symbols). Now we want to generalize well-known operations on sets to operations on models. For the special case where Σ is the empty signature, we obtain the analogous operation on sets.

10.1 Model Morphisms

10.1.1 The Analog for Sets: Functions

For any two sets M, N, we can form the set of functions $\varphi : M \to N$. Functions are essentially the most important concept of mathematics and computer science, representing structure and algorithms.

If M, N are Σ-models, the morphisms are functions $\varphi : \mathtt{term}^M \to \mathtt{term}^N$ that preserve the structure.

Example 10.1 (Monoid Morphisms). A morphism from a monoid $M_1 = (U_1, *_1, e_1)$ to a monoid $M_2 = (U_2, *_2, e_1)$ is a map $\varphi : U_1 \to U_2$ such that

- $\varphi(u *_1 v) = \varphi(u) *_2 \varphi(v)$,
- $\varphi(e_1) = e_2$.

φ preserves the operations: It maps composition to composition and unit to unit.

Example 10.2 (Monoid Morphisms). Consider $M_1 = (\mathbb{N}, +, 0)$ and $M_2 = (\mathbb{Z}, +, 0)$ and let $\varphi : \mathbb{N} \to \mathbb{Z}$ be the inclusion. It is easy to see that this inclusion is a monoid morphism.

For a less trivial example, consider M_1 and M_2 as before but let now $\varphi(u) = ku$ for some fixed $k \in \mathbb{Z}$. This is a monoid morphism for every k (even for $k = 0$).

For an example that relates very different monoids, consider $M_1 = (\Sigma^*, \cdot, \varepsilon)$ and $M_2 = (\mathbb{N}, +, 0)$ where Σ^* denotes the set of words over some fixed alphabet (i.e., Scala-strings), $+$ denotes the binary function that concatenates two words, and ε denotes the empty word. Then (M_1 is a monoid and) the length of a word is a monoid morphism from M_1 to M_2.

The above examples already indicate how powerful the concept of morphisms is: Relations between very different objects can be expressed succinctly. Furthermore, the definition of morphism does not depend so much on the particular structure that is supposed to be preserved. For monoids, the preserved structure consists of composition and unit.

[1] This chapter can be skipped.

10.1.2 Formal Definition

Definition 10.3 (Model Morphisms)**.** For two Σ-models M, N, a Σ-**morphism** $\varphi : M \to N$ is a mapping from $\mathtt{term}^M$ to $\mathtt{term}^N$ such that

- for every function symbol f or arity n and for all $u_1, \ldots, u_n \in \mathtt{term}^M$
$$\varphi(f^M(u_1, \ldots, u_n)) = f^N(\varphi(u_1), \ldots, \varphi(u_n))$$
- for every predicate symbol p or arity n and for all $u_1, \ldots, u_n \in \mathtt{term}^M$
$$p^M(u_1, \ldots, u_n) \leq p^N(\varphi(u_1), \ldots, \varphi(u_n))$$

φ is called **strong** if we have $=$ instead of $\leq$ in the cases for predicate symbol.

Example 10.4 (Morphisms). Morphisms come up virtually everywhere in mathematics.

- The models and morphisms of the empty theory are simply the sets and mappings.
- All inclusion maps between the number sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are morphisms for various signatures. For example, $(\mathbb{Z}, +, 0, -, \cdot, 1) \to (\mathbb{C}, +, 0, -, \cdot, 1)$ is a ring morphism.
- For any number set U and every $k \in U$, the mapping $M_k : x \mapsto kx$ (which multiplies by k) is a monoid morphism $(U, +, 0) \to (U, +, 0)$ because
$$k(x + y) = kx + ky \qquad \text{and} \qquad k0 = 0.$$
- The exponential function $\exp : x \mapsto e^x$ is a group morphism $(\mathbb{R}, +, 0, -) \to (\mathbb{R}^+, \cdot, 1, ^{-1})$ where $\mathbb{R}^+$ is the set of positive real numbers because
$$e^{x+y} = e^x \cdot e^y \qquad \text{and} \qquad e^0 = 1 \qquad \text{and} \qquad e^{-x} = 1/(e^x).$$
Its inverse ln is a group morphism in the opposite direction.
- Let $\mathit{modulus}_k : \mathbb{Z} \to \mathbb{Z}_k$ be the mapping $n \mapsto n \bmod k$. Then $\mathit{modulus}_k$ is a ring morphism $(\mathbb{Z}, +, 0, -, \cdot, 1) \to (\mathbb{Z}_k, +_k, 0, -_k, \cdot_k, 1)$. Moreover, if $k|l$, the mapping $\mathit{modulus}_k : \mathbb{Z}_l \to \mathbb{Z}_k$ is a ring morphism between the corresponding rings. (The former is a special case of the latter using $l = 0$, $k|0$, and $\mathbb{Z}_0 = \mathbb{Z}$.)
- The real and imaginary part functions Re, Im are homomorphisms $(\mathbb{C}, +, 0, -) \to (\mathbb{R}, +, 0, -)$ because
$$\mathrm{Re}(x + y) = \mathrm{Re}x + \mathrm{Re}y \qquad \text{and} \qquad \mathrm{Re}0 = 0 \qquad \text{and} \qquad \mathrm{Re}(-x) = -\mathrm{Re}x$$
and accordingly for Im.
- The conjugation $x + yi \mapsto x - yi$ is a field isomorphism from $(\mathbb{C}, +, 0, -, \cdot, 1, ^{-1})$ to itself.
- The order morphisms $\varphi : (U, \leq) \to (U', \leq')$ are the monotonous mappings, i.e., the mappings for which $u \leq v$ implies $\varphi(u) \leq' \varphi(v)$.
- The complement $-^C : \mathcal{P}(S) \to \mathcal{P}(S)$ is a lattice morphism $(\mathcal{P}(S), \cup, \varnothing, \cap, S) \to (\mathcal{P}(S), \cap, S, \cup, \varnothing)$. It is also a strong order morphism $(\mathcal{P}(S), \subseteq) \to (\mathcal{P}(S), \supseteq)$.
- The morphisms between vector spaces are the linear maps.
- The length-function $|-|$ is a monoid morphism $(\Sigma^*, \cdot, \varepsilon) \to (\mathbb{N}, +, 0)$ because
$$|vw| = |v| + |w| \qquad \text{and} \qquad |\varepsilon| = 0.$$
- Let U be a sufficiently large set of finite sets that contains $0 := \varnothing$, $1 := \{\varnothing\}$, disjoint unions $A \sqcup B := \{0\} \times A \cup \{1\} \times B$, and function sets $A \to B$. Then the cardinality-function $|-|$ is a morphism $(U, \sqcup, \varnothing, \times, \{\varnothing\}, \to) \to (\mathbb{N}, +, 0, \cdot, 1, (x, y) \mapsto y^x)$ because
$$|\varnothing| = 0 \qquad |\{\varnothing\}| = 1 \qquad |A \sqcup B| = |A| + |B| \qquad |A \times B| = |A| \cdot |B| \qquad |A \to B| = |B|^{|A|}.$$

10.1.3 Morphisms as Property-Preserving Maps

Intuitively, morphisms preserve properties. In fact, the main reason why morphisms are interesting is that they preserve properties – many important mathematical theorems can be stated as the preservation of some property under some operation. The following definition makes precise what we mean by that:

Definition 10.5 (Preservation and Reflection)**.** Consider a morphism $\varphi : M \to N$ and a Γ-formula F.
If α is an assignment from Γ-into M, we write $\varphi(\alpha)$ for the assignment from Γ into N that maps $\varphi(\alpha)(x) = \varphi(\alpha(x))$.
Then we say that

- φ **preserves** F if for every assignment α

$$[\![F]\!]^{M,\alpha} \leq [\![F]\!]^{N,\varphi(\alpha)}$$

- φ **reflects** F if for every assignment α

$$[\![F]\!]^{M,\alpha} \geq [\![F]\!]^{N,\varphi(\alpha)}$$

For a Γ-term t, we say that φ preserves/reflects t if it preserves/reflects the formula $y = t$, where y is an additional variable that is not in Γ.
We say that φ preserves/reflects the predicate symbol p if it preserves/reflects the formula $p(x_1, \ldots, x_n)$. We say that φ preserves/reflects the function symbol f if it preserves/reflects the term $f(x_1, \ldots, x_n)$.

Note that the definition of the preservation of F can be rephrased more intuitively as: if F holds in M under α, then it holds in N under $\varphi(\alpha)$. Even more intuitively, F stays true when going from M to N via φ. That is why we call it *preservation*. The definition of reflection can be rephrased accordingly.
Moreover, we can rephrase the definition of morphisms as follows: A mapping φ is a morphism iff it preserves function and predicate symbols. A morphism reflects function symbols if it is injective and reflects predicate symbols iff it is strong. In fact, morphisms preserve even more:

Theorem 10.6 (Preservation)**.** *A morphism preserves all terms and all atomic formulas.*

Proof. Exercise. □

Morphisms do *not* in general preserve *all* formulas. The preservation of all formulas is usually only possible if the involved models are very similar. But morphisms are most interesting and valuable if they allow connecting very different models, i.e., models that satisfy very different formulas. Therefore, morphisms are only required to preserve atomic formulas.

Theorem 10.7 (Preservation/Reflection of Complex Formulas)**.** *Under the following conditions, a morphism φ preserves/reflects the respective formula:*

The formula	*is preserved if*	*is reflected if*
$s \doteq t$	*always*	φ *injective*
$p(t_1, \ldots, t_n)$	*always*	φ *strong*
true	*always*	*always*
false	*always*	*always*
$\neg F$	F *reflected*	F *preserved*
$F \wedge G$	F, G *preserved*	F, G *reflected*
$F \vee G$	F, G *preserved*	F, G *reflected*
$F \Rightarrow G$	F *reflected,* G *preserved*	F *preserved,* G *reflected*
$\forall x.\ F$	F *preserved,* φ *surjective*	F *reflected*
$\exists x.\ F$	F *preserved*	F *reflected,* φ *surjective*

Proof. Exercise. □

For example, we can see from the table that

- morphisms preserve all formulas that do not use $\forall$, $\neg$, or $\Rightarrow$, and $\forall$ is allowed in the surjective case,
- injective strong morphisms reflect all formulas that do not use $\exists$, $\neg$, or $\Rightarrow$,
- bijective strong morphisms preserve and reflect all formulas.

10.1.4 Special Morphisms

Theorem 10.8 (Identity Morphism). *For every theory and every model M, the identity $id_M : M \to M$, which maps $id_M(x) = x$, is a morphism.*

Proof. Exercise. □

Theorem 10.9 (Composition). *For every theory and morphisms $\varphi_1 : M_1 \to M_2$ and $\varphi_2 : M_2 \to M_3$, the composition mapping $\varphi_1;\varphi_2 : M_1 \to M_3$, which maps $(\varphi_1;\varphi_2)(x) = \varphi_2(\varphi_1(x))$, is a morphism.*

Proof. Exercise. □

Definition 10.10 (Powers). Given a model morphism $\varphi : M \to M$, we define $\varphi^n : M \to M$ by

$$\varphi^0 = id_M \qquad \text{and} \qquad \varphi^{n+1} = \varphi^n;\varphi$$

In particular, $\varphi^1 = \varphi$ and $\varphi^{n+1}(x) = \varphi(\varphi^n(x))$.

Theorem 10.11 (Kinds of Morphisms). *Consider a morphism $\varphi : M \to N$. Then:*

- *The following are equivalent:*
  - *φ is an injective mapping*
  - *φ can be canceled in terminal position: $\psi;\varphi = \psi';\varphi$ implies $\psi = \psi'$*

  In that case, φ is called a ***mono(morphism)****.*
- *The following are equivalent:*
  - *φ is a surjective mapping*
  - *φ can be canceled in initial position: $\varphi;\psi = \varphi;\psi'$ implies $\psi = \psi'$*

  In that case, φ is called an ***epi(morphism)****.*
- *The following are equivalent:*
  - *φ is strong and a bijective mapping*
  - *φ is strong, mono, and epi*
  - *φ can be inverted: there is a morphism φ^{-1} such that $\varphi;\varphi^{-1} = id_M$ and $\varphi^{-1};\varphi = id_N$*

  In that case, φ is called an ***iso(morphism)****.*

Proof. Exercise. For isomorphisms, note that the mapping φ^{-1} exists *uniquely* because the mapping φ is bijective. □

Theorem 10.12. *The identity is iso and thus in particular strong, mono, and epi.*

If two morphisms are strong, mono, epi, or iso, then so is their composition.

The inverse of an iso is also iso.

Proof. Exercise. □

Terminology 10.13. Morphism from $M \to M$ are sometimes called **endomorphisms** of M. Iso-Endo-Morphisms of M are sometimes called **automorphisms** of M.

10.2 Closed Subsets and Submodels

10.2.1 The Analog for Sets: Subsets

If for every $x \in M$, we also have $x \in N$, we write $M \subseteq N$ and call M a subset of N. In that case, we have the inclusion function $M \hookrightarrow N$, which maps every $x \in M$ to itself.

10.2.2 Formal Definition

Closed Subsets

Definition 10.14 (Closed Set)**.** Consider a Σ-model N. A subset $S \subseteq \texttt{term}^N$ is called **closed** if

- for every n-ary function symbol f

$$\text{if} \qquad u_1 \in S, \dots, u_n \in S \qquad \text{then} \qquad f^M(u_1, \dots, u_n) \in S.$$

Images of Morphisms

Definition 10.15 (Image)**.** Consider a Σ-morphism $\varphi : M \to N$. The **image** $\operatorname{im}\varphi$ of φ is the subset of N defined by $\{\varphi(m) : m \in \texttt{term}^M\}$.

Theorem 10.16. *The image of a morphism is closed.*

Proof. Exercise. □

Submodels

Definition 10.17 (Submodel)**.** Consider a Σ-model N and a closed set S. We define the submodel $M = N|S$ by

- $\texttt{term}^M = S$,
- for every n-ary function symbol f

$$f^M(u_1, \dots, u_1) = f^N(u_1, \dots, u_n),$$

- for every n-ary predicate symbol p

$$p^M(u_1, \dots, u_1) = p^M(u_1, \dots, u_n).$$

Well-definedness: f^M is well-defined (i.e., returns an element of $\texttt{term}^M$) because S is closed.

Intuitively, the submodels of N are the subsets of $\texttt{term}^N$ that are closed.

Theorem 10.18. *The submodels of N are exactly those models M with $\texttt{term}^M \subseteq \texttt{term}^N$ for which the inclusion mapping is a morphism.*

Proof. Consider a subset S of $\texttt{term}^N$. We have the inclusion mapping $S \to \texttt{term}^N$, which maps $x \mapsto x$. We have to show that the following are equivalent:

1. S is closed.
2. The inclusion is a morphism $N|S \to N$

This is an exercise. □

Example 10.19 (Submodels). Almost every model has a rich structure of different submodels.

- The submodels of the empty theory are simply the subsets.
- Every model M is its own submodel $M \leq M$. Of course, this is the greatest possible submodel of M.
- For every model M, we can form the set $\mathit{Defin}_M := \{u \in \texttt{term}^M \mid \text{exists } t \text{ such that } [\![t]\!]^M = u\}$. Intuitively, it contains all definable elements, i.e., all elements that can be named by a term t in the syntax. Defin_M is the smallest submodel of M. For example, $\mathit{Defin}_O = \varnothing$ for every order O, and $\mathit{Defin}_R = \{0\}$ for every ring R.
- All inclusion maps between the number sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ yield submodels for various signatures. For example, $(\mathbb{Z}, +, 0, -, *, 1)$ is a subring of $(\mathbb{C}, +, 0, -, *, 1)$. And the rational numbers are a subfield of the real numbers, which are a subfield of the complex numbers.

- The positive integers with 0 form a subsemiring of $(\mathbb{Z}, +, 0, \cdot, 1)$. They do not form a subring because they are not closed under negation. The according result applies to rational and real numbers.
- The positive real numbers without 0 form a subgroup of $(\mathbb{R}, \times, 1, ^{-1})$. The according result applies to rational numbers.
- For $k \in \mathbb{N}$, let $k\mathbb{Z} = \{kz : z \in \mathbb{Z}\} = \{z \in \mathbb{Z} \mid k|z\}$ be the set of integers that are divisible by k. (Note that $1\mathbb{Z} = \mathbb{Z}$ and $0\mathbb{Z} = \{0\}$.) This set is the image of the endomorphism $z \mapsto kz$. Thus $k\mathbb{Z}$ is closed and yields a subring of the integers. Moreover, we have $k\mathbb{Z} \subseteq l\mathbb{Z}$ iff $l|k$.
- The universal relation $\top_M = \{(x, y) : x, y \in \mathtt{term}^M\} = \mathtt{term}^M \times \mathtt{term}^M$ is an equivalence relation (and thus a model of the theory of equivalence relations). Its submodels are the equivalence relations on $\mathtt{term}^M$.

10.2.3 Interaction with Theories

All of the above has been formulated for *signatures* Σ, not for *theories* $(\Sigma; \Theta)$. This is because if M and N are Σ-models and $M \leq N$, and N is a $(\Sigma; \Theta)$-model, then M is not necessarily also $(\Sigma; \Theta)$-model. Similarly, if M is a $(\Sigma; \Theta)$-models, N may or may not be. In other words, if N as a $(\Sigma; \Theta)$, it has fewer submodels if Θ has more axioms.

These negative results are a consequence of the fact that morphisms do not preserve/reflect all formulas. However, we do not have an arbitrary morphism here: we have an inclusion morphism $M \to N$, and we know that inclusion morphisms are strong and mono. Then Thm. 10.7 tells us that a propositional formula (i.e., a formula without $\forall$, $\exists$) holds in M for some assignment iff it holds in N for the same assignment.

Therefore, we have at least some positive results: Universally quantified propositional formulas that hold in a model also hold in any submodel; thus, if all axioms in Θ have that form, any Σ-submodel is also a $(\Sigma; \Theta)$ submodel. Similarly, existentially quantified propositional formulas that hold in a model also hold in any supermodel. thus, if all axioms in Θ have that form, any Σ-supermodel is also a $(\Sigma; \Theta)$ supermodel.

Especially the former case is important in practice: Many important theories only use axioms that are universally quantified propositional formulas, including most of the theories from Sect. 3.6.2.

10.3 Product Models

10.3.1 The Analog for Sets: Cartesian Product

For any two sets M_1, M_2, we can form the set $M_1 \times M_2$. It comes with projection functions $(-)_i : M_1 \times M_2 \to M_i$ such that $(u, v)_1 = u$ and $(u, v)_2 = v$.

10.3.2 Formal Definition

The product of two models uses the cartesian product of the universes as its universe and interprets every function and predicate symbol component-wise:

Definition 10.20 (Product Model)**.** Consider two Σ-models M_1, M_2. We define the Σ-model $M_1 \times M_2$ as follows:

- $\mathtt{term}^{M_1 \times M_2} = \mathtt{term}^{M_1} \times \mathtt{term}^{M_2}$,
- for every n-ary function symbol f

$$f^{M_1 \times M_2}(u^1, \ldots, u^n) = (f^{M_1}(u^1_1, \ldots, u^n_1), f^{M_2}(u^1_2, \ldots, u^n_2))$$

- for every n-ary predicate symbol p

$$p^{M_1 \times M_2}(u^1, \ldots, u^n) = 1 \qquad \text{iff} \qquad p^{M_1}(u^1_1, \ldots, u^n_1) = 1 \text{ and } p^{M_2}(u^1_2, \ldots, u^n_2) = 1$$

For $i = 1, 2$, we obtain projection morphisms $(-)_i : M_1 \times M_2 \to M$ that map $(u, v)_1 = u$ and $(u, v)_2 = v$.

Example 10.21. Products are one the most important ways to build larger models from smaller ones. (Submodels and quotients only produce smaller models from larger ones.)

- The commutative group $(\mathbb{C}, +, 0, -)$ can be seen as the product of $(\mathbb{R}, +, 0, -)$ with itself because

$$(x+yi)+(x'+y'i) = (x+x')+(y+y')i \qquad 0 = 0+0i \qquad -(x+yi) = -x+(-y)i$$

  The two projections are Re and Im.
  However, complex multiplication cannot be obtained in this way.
- Every k-dimensional F-vector space is isomorphic to the n-th power P (i.e., the n-fold product with itself) of $(F, +, 0, -)$. The operations in P are

$$\begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix} +^P \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} u_1+v_1 \\ \vdots \\ u_n+v_n \end{pmatrix} \qquad 0^P = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \qquad -^P \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} -u_1 \\ \vdots \\ -u_n \end{pmatrix}$$

- The product of two rings is again a ring.
- The product $P = F \times G$ of two fields is only a ring and not a field. The problem is that elements like $(0^F, 1^G)$ are different from $0^P = (0^F, 0^G)$ but do not have a multiplicative inverse.
- In the product $P = O \times O'$ of two orders, pairs $(u, u') \leq^P (v, v')$ if $u \leq^O v$ and $u' \leq^{O'} v'$.

10.3.3 Interaction with Theories

If M_1 and M_2 are $(\Sigma; \Theta)$-models, then $M_1 \times M_2$ may or may not be a $(\Sigma; \Theta)$-model. Therefore, if we restrict attention to $(\Sigma; \Theta)$-models we may not always have product models.

However, we can apply Thm. 10.7 using that the projection morphisms are epi (but not necessarily strong) if both factors are non-empty. Moreover, we know that $(u, v) = (u', v')$ iff $u = u'$ and $v = v'$. Therefore, if both factors are non-empty, then any formula that does not use $\neg$, $\vee$, or $\Rightarrow$ holds in a product iff it holds in both factors. Similarly, any Horn formula holds in a product if it holds in both factors.

Thus, many important theories admit product models, including most of the ones from Sect. 3.6.2. The most important exception is the theory of fields (which uses an axiom with $\neg$).

10.4 Power Models

10.4.1 The Analog for Sets: Families of Elements

For any two sets M, I, we can form the set M^I of functions from M to I. Such a function can be seen as a family $(u_i)_{i \in I}$ of values $u_i \in M$.

10.4.2 Formal Definition

We obtain a simple generalization to morphisms if we use a Σ-model M and a set I. We call I the index set.

Definition 10.22 (Power Model)**.** Consider a Σ-model M and a set I. We define the Σ-model $P = M^I$ as follows:
- $\mathtt{term}^P = (\mathtt{term}^M)^I$,
- for every n-ary function symbol f

$$f^P(u^1, \ldots, u^n) = (f^M(u^1_i, \ldots, u^n_i))_{i \in I}$$

- for every n-ary predicate symbol p

$$p^P(u^1, \ldots, u^n) = 1 \qquad \text{iff} \qquad p^M(u^1_i, \ldots, u^n_i) = 1 \text{ for all } i \in I$$

For every $i \in I$, we obtain a projection morphism $M^I \to M$ that maps $u \mapsto u_i$.

Example 10.23. Power models have some important cases if I is finite:

- If I is the set $\{0, \ldots, k-1\}$, then M^I is isomorphic to the k-th power of, i.e., $M \times \ldots \times M$. In particular, $M^{\{0,1\}}$ is isomorphic to $M \times M$, and the two definitions of the projection morphisms coincide.
- $M^{\{0\}}$ is isomorphic to M itself, and the projection is that isomorphism.
- $M^{\varnothing}$ is isomorphic to the trivial model, and the projection is the unique morphism into the terminal model (see Sect. 10.5).

10.4.3 Interaction with Theories

If M is a $(\Sigma;\Theta)$-model, then M^I may or may not be a $(\Sigma;\Theta)$-model. Therefore, if we restrict attention to $(\Sigma;\Theta)$-models we may not always have power models.

However, we can use the same arguments as for product models to show that M^I is a $(\Sigma;\Theta)$-model for many important special cases of Θ.

10.5 Terminal Models

Terminal models do not always exist and are not always useful. But they are the dual of the crucially important concept of initial models (see Sect. 10.6).

10.5.1 The Analog for Sets: The Trivial Sets

Singleton sets (i.e., sets of size 1) have a special property. If $T = \{t\}$ for some t, there is a unique mapping $M \to T$ for every set M: It maps every $x \in M$ to t.

10.5.2 Formal Definition

Definition 10.24 (Terminal Model)**.** A $(\Sigma;\Theta)$-model T is called **terminal** if for every $(\Sigma;\Theta)$-model M there is a unique morphism $M \to T$.

Theorem 10.25 (Trivial Model)**.** *For $\Theta = \varnothing$, we obtain a terminal Σ-model T as follows:*

- $\mathtt{term}^T$ *is some singleton set, e.g.,* $\{*\}$,
- f^T *for function symbols T is the uniquely determined map that always returns* $*$,
- p^T *for predicate symbols p is the map that always returns* 1.

Moreover, all terminal Σ-models are of this form.

Proof. Exercise. □

10.5.3 Interaction with Theories

If $\Theta \neq \varnothing$, the terminal Σ-model may or may not be a $(\Sigma;\Theta)$-model. If it is a $(\Sigma;\Theta)$-model, it is also a terminal $(\Sigma;\Theta)$-model. If not, there may or not be other $(\Sigma;\Theta)$-models that are terminal.

However, it is easy to see that the terminal Σ-model satisfies all formulas that do not contain *false* or $\neg$. Therefore, it is terminal for many practically important theories. Again, fields are the most important exception.

10.6 Initial Models

10.6.1 The Analog for Sets: The Smallest Set

The empty set $\varnothing$ is the smallest set in the sense that $\varnothing \subseteq M$ for every set M. Moreover, for every set M, there is a unique map $\varnothing \to M$, namely the empty map $\varnothing : \varnothing \hookrightarrow M$.

10.6.2 Formal Definition

We do not always have a smallest Σ-model. But we do have the dual of the terminal model:

Definition 10.26 (Initial Model)**.** A Σ-model T is called **initial** if for every Σ-model M there is a unique morphism $I \to M$.

Theorem 10.27 (Term Model)**.** *For $\Theta = \varnothing$, we obtain an initial Σ-model I as follows:*

- $\mathtt{term}^I$ *is the set of all closed Σ-terms,*
- f^I *for function symbols T maps $f^I(t_1, \dots, t_n) = f(t_1, \dots, t_n)$,*
- p^I *for predicate symbols p always returns 0.*

Proof. The unique morphism $u : I \to M$ maps $u(t) = [\![t]\!]^M$. The rest is left as an exercise. □

$\mathcal{FOL}$ does not have an initial model for every theory. The most important theories for which initial models exists are the Horn theories:

Theorem 10.28 (Term Model for Horn Theories)**.** *If Θ contains only Horn formulas, we obtain an initial Σ-model I as follows:*

- $\mathtt{term}^I$ *is the set of all closed Σ-terms quotiented by the equivalence relation $\equiv$ defined by*
$$s \equiv t \qquad \text{iff} \qquad \Theta \vdash_\Sigma s \doteq t.$$
- f^I *for function symbols T maps*
$$f^I([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)],$$
- p^I *for predicate symbols p maps*
$$p^I([t_1], \dots, [t_n]) = \begin{cases} 1 & \text{if } \Theta \vdash_\Sigma p(t_1, \dots, t_n) \\ 0 & \text{otherwise} \end{cases}$$

Proof. The model is well-defined because $\equiv$ is a congruence relation (see Sect. 10.7). Moreover, I is the quotient of the term model constructed in Thm. 10.27 by $\equiv$.
The unique morphism $u : I \to M$ maps $u([t]) = [\![t]\!]^M$. The rest is left as an exercise. □

Note that the term model of Thm. 10.27 is isomorphic to the one from Thm. 10.28 for $\Theta = \varnothing$.

Example 10.29 (Initial Models). Many data types can be specified by Horn theories, and we obtain many initial models:

- The initial set is the empty set.
- More generally, the empty model is initial whenever it is a model at all (i.e., whenever it satisfies all axioms in Θ), e.g., for the theory of orders.
- The trivial model is initial for monoids, groups, rings without 1, and vector spaces. Indeed, we can check that all closed terms over these signatures are equal to the neutral element.
- The integers are an initial ring with 1. Indeed, the smallest set of not-provably-equal terms over the theory of rings with 1 is $\{0, 1, -1, 1+1, -(1+1), \dots\}$.
- Peano arithmetic from Ex. 1.35 is not a Horn theory. However, the term model is still initial. It is isomorphic to the natural numbers. Indeed, the smallest set of not–provably equal terms over Peano arithmetic is $\{0, succ(0), succ(succ(0)), \dots\}$.
- The theory of fields is not a Horn theory and has no initial model.

10.6.3 Interaction with Theories

The initial $(\Sigma; \Theta)$-model I from Thm. 10.28 has a very special property — it satisfies exactly the theorems:

$$[\![F]\!]^I = 1 \qquad \text{iff} \qquad \Theta \vdash_\Sigma F$$

For atomic formulas, that holds by construction. For the remaining formulas, we can show it by induction using the fact all axioms are Horn formulas.

If Θ contains non-Horn formulas, I is usually not a $(\Sigma;\Theta)$-model and therefore also not an initial $(\Sigma;\Theta)$-model. However, there may be other $(\Sigma;\Theta)$-models that are initial.

10.7 Congruence Relations and Quotient Models

As we will see in Sect. 10.9.4, quotients are the dual construction to submodels.

10.7.1 The Analog for Sets: Equivalence Relations and Quotient Sets

An equivalence relation on a set M is a binary relation $\equiv$ on M that is reflexive, symmetric, and transitive. In that case, we can form the equivalence class $[m] := \{x \in M \,|\, m \equiv x\}$ for every $m \in M$.

The set $M/\equiv := \{[m] : m \in M\}$ of all equivalence classes is called the quotient of M by $\equiv$.

The classification mapping $[-] : M \to M/\equiv$ maps every m to its equivalence class. It is surjective, and we have $m \equiv n \iff [m] = [n]$.

10.7.2 Formal Definition

Congruences

It is easy to obtain equivalence relations on models. But not every equivalence relation interacts well with the symbols. We are interested in those relations that are closed under the symbols:

Definition 10.30 (Congruence Relation)**.** Consider a Σ-model M and a binary relation r on M. r is called a **pre-congruence** if

- for every n-ary function symbol f

$$\text{if} \quad (u_1, v_1) \in r, \ \dots, (u_n, v_n) \in r \quad \text{then} \quad (f^M(u_1, \dots, u_n), f^M(v_1, \dots, v_n)) \in r$$

- for every n-ary predicate symbol f

$$\text{if} \quad (u_1, v_1) \in r, \ \dots, (u_n, v_n) \in r \quad \text{then} \quad p^M(u_1, \dots, u_n) = p^M(v_1, \dots, v_n)$$

r is called a **congruence** if it is additionally an equivalence relation.

Kernels of Morphisms

Definition 10.31 (Kernels)**.** Consider a Σ-morphism $\varphi : M \to N$. The kernel $\ker\varphi$ of φ is the binary relation on M defined by $\{(x, y) \in \mathtt{term}^M \times \mathtt{term}^M \mid \varphi(x) = \varphi(y)\}$.

Theorem 10.32. *The kernel of a morphism is a congruence.*

Proof. Exercise. □

Quotients

Definition 10.33 (Quotient Model)**.** Consider a Σ-model M and a congruence $\equiv$ on it. We define the Σ-model $Q = M/\equiv$ as follows:

- $\mathtt{term}^Q = \mathtt{term}^M/\equiv$,
- for every n-ary function symbol f

$$f^Q([u_1], \dots, [u_1]) = [f^M(u_1, \dots, u_n)],$$

- for every n-ary predicate symbol p

$$p^Q([u_1], \ldots, [u_1]) = p^M(u_1, \ldots, u_n).$$

Well-definedness: Here the arguments of f^Q are equivalence classes, and we define the result of f^Q by applying f^M to arbitrary elements u_i of these equivalence classes. This is well-defined because the properties of pre-congruences guarantee that all values $f^M(u_1, \ldots, u_n)$ are in the same equivalence class, no matter which u_i we pick. The argument that p^Q is well-defined proceeds accordingly.

Theorem 10.34. *The quotients of M are exactly those models with universe $\texttt{term}^M/\equiv$ for which the classification mapping is a morphism.*

Proof. Consider an equivalence relation $\equiv$ on $\texttt{term}^M$. We have the mapping $[-] : \texttt{term}^M \to \texttt{term}^M/\equiv$, which maps $x \mapsto [x]$. We have to show that the following are equivalent:

1. $\equiv$ is a congruence.
2. $[-] : M \to M/\equiv$ is a morphism.

This is an exercise. □

Example 10.35. Almost every model has a rich structure of quotient models:

- Modulo k: The relation $m \equiv_k n$ on $\mathbb{Z}$ is defined by $m - n \in k\mathbb{Z}$. It is a congruence on the ring with 1 of integers. It is the kernel of the morphism $modulus_k : \mathbb{Z} \to \mathbb{Z}_k$. $\mathbb{Z}_k$ is isomorphic to the corresponding quotient.
- A group is called simple if it has no non-trivial quotient groups. In a huge collaborative undertaking, mathematicians have been able to find all finite simple groups [**?**], one of the biggest results ever in mathematics.
- The diagonal relation $\Delta_M = \{(x, x) : x \in \texttt{term}^M\}$ is a congruence (moreover, the smallest possible congruence). The quotient M/Δ_M is isomorphic to M.
- The universal relation $\top_M$ is a congruence (moreover, the largest possible congruence). The quotient $M/\top_M$ is a trivial model.

10.7.3 Interaction with Theories

If M is $(\Sigma; \Theta)$-model, then $M/\equiv$ may or not be a $(\Sigma; \Theta)$-model. Thus, M has fewer quotients if Θ has more axioms.

However, we know that the classification morphism is strong and epi. Therefore, Thm. 10.7 tells us that any formula that does not use $\Rightarrow$ or $\neg$ that holds in a model also holds in every quotient.

10.8 Subquotients

10.8.1 The Analog for Sets: Partial Equivalence Relations

A binary relation $\equiv$ on a set M is a **partial equivalence** relation (PER) if it is symmetric and transitive.

Contrary to an equivalence relation, a PER need not be reflexive. However, it is easy to see that if $\equiv$ is a PER, then $x \equiv x$ for every $x \in M$ for which $x \equiv y$ or $y \equiv x$ for any $y \in M$. Thus, the only counter-examples to reflexivity are those x that are not in relation with any other element. Formally, we have: A PER on M is exactly the same as an equivalence relation on a subset of M.

PERs allow combining the subset and the quotient operation in one go. Moreover, they contain subsets and quotients as special case: If $\equiv \subseteq \Delta_M$, then $\equiv$ identifies a subset, namely the subset containing those $x \in M$ for which $x \equiv x$. If $\equiv \supseteq \Delta_M$, then $\equiv$ is reflexive and thus an equivalence relation and identifies a quotient.

10.8.2 Formal Definition

It is straightforward to define the subquotients of a model M as the quotient models of the submodels of M. But here we will give a more general definition: A subquotient of M is any model m that is *isomorphic to* a quotient of a submodel of M. This allows m to exist independently of M, which is often more practical.

Definition 10.36 (Subquotient). Consider two Σ-models M and m and a relation $\sim \subseteq \mathtt{term}^m \times \mathtt{term}^M$. $(m, \sim)$ is called a **subquotient** of M if

- $\sim$ is total and injective (equivalently: $\sim^{-1}$ is functional and surjective),
- for every n-ary function symbol f

$$\text{if} \quad x_1 \sim X_1, \ldots, x_n \sim X_n \quad \text{then} \quad f^m(x_1, \ldots, x_n) \sim f^M(X_1, \ldots, X_n)$$

  for all $x_i \in \mathtt{term}^m$ and $X_i \in \mathtt{term}^M$.

Neither $\sim$ nor $\sim^{-1}$ is a function. But when implementing subquotients in computers, it is nicer to have only functions. Therefore, we use them like functions as follows:

- We write r if we use $\sim^{-1}$ as a function: $r(X) = x$ iff $x \sim X$. There is at most one such x but there may be none, i.e., r is a partial function $\mathtt{term}^M \to \mathtt{term}^m$.
- We write l if we use $\sim$ as a function: $l(x) = X$ iff $x \sim X$. There is at least one such X but there may be multiple, i.e., l is a non-deterministic function $\mathtt{term}^m \to \mathtt{term}^M$ (which randomly returns one such X).

The functions l and r are often called **lift** and **retract**. The condition that $\sim$ is total and injective means that $lr = \Delta_{\mathtt{term}^m}$, i.e., that retraction undoes the effect of lifting. So we can lift elements of the subquotient m to the base model M, and retraction pulls them back from M to m.

We will look at the special cases first: For submodels, l is the embedding of the submodel m into M, and r is the partial function that undoes the embedding. For quotients, l is the representative-selection that maps each equivalence class to any one of its elements, and r undoes the representative-selection by projection all elements to their equivalence class. More precisely, we have:

Theorem 10.37. *Consider a subquotient $(m, \sim)$ of M.*
If $\sim$ is additionally functional,

- *$\sim$ and thus l are functions,*
- *m is isomorphic to the submodel S of M given by the image of l,*
- *l is the isomorphism $m \to S$, and the restriction of r to the image of l is the inverse isomorphism $S \to m$.*

If $\sim$ is additionally surjective, then

- *$\sim^{-1}$ and thus r are functions,*
- *m is isomorphic to the quotient Q of M by the kernel of r,*
- *The map $[X]_Q \mapsto r(X)$ is the isomorphism $Q \to m$, and the map $x \mapsto [l(x)]_Q$ is the inverse isomorphism $m \to Q$.*

If $\sim$ is both functional and surjective, then

- *l and r are functions,*
- *m is isomorphic to M,*
- *$l : m \to M$ and $r : M \to m$ are the isomorphisms.*

Now we state the general case:

Theorem 10.38. *Consider a subquotient $(m, \sim)$ of M. Let*

$$s = \{X \in \mathtt{term}^M \mid x \sim X \text{ for some } x \in \mathtt{term}^m\}$$

(i.e., s is the image of l). Then s is closed (wrt to all Σ-function symbols), and we define $S = M|s$. Now let

$$q = \{(X, X') \in \mathtt{term}^M \times \mathtt{term}^M \mid x \sim X \text{ and } x \sim X' \text{ for some } x \in \mathtt{term}^m\}$$

(i.e., q is the kernel of r). Then q is a congruence (wrt to all Σ-function symbols), and we define $Q = S/q$.
Then

- *m is isomorphic to Q,*
- *The map $[X]_Q \mapsto r(X)$ is the isomorphism $Q \to m$, and the map $x \mapsto [l(x)]_Q$ is inverse isomorphism $m \to Q$.*

Above, we only defined what it means for a model m to be a subquotient of M. But in practice, it is often more important to construct m from M if l and r are given. We have the following:

Theorem 10.39. *Consider a Σ-model M and a set $\texttt{term}^m$, a relations $l \subseteq \texttt{term}^m \times \texttt{term}^M$ and $r \subseteq \texttt{term}^M \times \texttt{term}^m$ such that $lr = id_{\texttt{term}^m}$ Then l is a non-deterministic function $l : \texttt{term}^m \to \texttt{term}^M$ and r is a partial function $r : \texttt{term}^m \to \texttt{term}^M$.*
Assume that r is defined on the closure of the image of l in M.
Then it is well-defined to extend m to a Σ-model by

$$f^m(x_1, \dots, x_n) = r(f^M(l(x_1), \dots, l(x_n))$$

And $(m, \sim)$ is a subquotient of M for some relation $\sim$ that satisfies $l \subseteq \sim \subseteq r^{-1}$.

Proof. $\sim$ is obtained by taking r^{-1} restricted to the closure of the image of l. □

Because we can implement l and r as (non-deterministic or partial) functions in a programming language, they can be much easier to work with than the relation $\sim$. In this case, we can think of l and r as under- and over-approximations of $\sim$. Knowing that m is a subquotient and being able to compute in m (for which we need only l and r but not $\sim$) is usually sufficient in practice.

10.9 Meta-Mathematics

10.9.1 Isomorphism as an Equivalence Relation

The relation $M \cong N$ defined by "there is an isomorphism from M to N" is an equivalence relation. Thus, we can quotient $\mathbf{Mod}(\Sigma; \Theta)$ by $\cong$.
For a property P, we say "P holds up to isomorphism" if the following holds: If P holds for M and $M \cong N$, then P also holds for N. In other words, P holds in the quotient $\mathbf{Mod}(\Sigma; \Theta)/\cong$.

10.9.2 Models Containing Morphisms

Given a model M, the endomorphisms of M can be composed with each other. Thus, they form a magma $(Endo(M), \circ)$.
Moreover,

- $(Endo(M), \circ, id_M)$ is a monoid.
- The set of automorphisms of M is a closed subset of $(Endo(M), \circ, id_M)$. In other words, the identity is an automorphism, and the composition of automorphisms is an automorphism.) $(Auto(M), \circ, id_M, inv)$ is a group.

10.9.3 Product as a Monoid

The product model operation $M \times N$ is a magma on $\mathbf{Mod}(\Sigma; \Theta)$. $\cong$ is a congruence with respect to $\times$ and thus the quotient $\mathbf{Mod}(\Sigma; \Theta)/\cong$ is also a magma. This quotient has several nice properties:

- It is commutative: $M \times N \cong N \times M$.
- It is associative.
- The trivial Σ-model (if it is Θ-model) is a neutral element.
- The empty model (if it is a Θ-model) is an absorbing element.

Here it is necessary to work "up to isomorphism", i.e., in the quotient by $\cong$. For example, we have $M \times N \cong N \times M$ but not $M \times N = N \times M$. Similarly, we have $A \times \{*\} \cong A$ but not $A \times \{*\} = A$. Thus, the axioms do not hold exactly — they only hold if we identify isomorphic models. An exception is the absorbing element: We do have $A \times \varnothing = \varnothing$, not only $A \times \varnothing \cong \varnothing$.

10.9.4 Duality

Terminal and Initial Models

Inspecting the definitions of terminal and initial models, we can spot an important duality: The definitions are exactly the same except for flipping the direction of the arrows. Initial models have unique morphisms *into* any other models, and terminal models have unique morphisms *out of* any other model.

Another duality between the two is that both are unique up to isomorphism: any two initial models of the same theory are isomorphic, and any two terminal models of the same theory are isomorphic.

This is a special case of an extremely powerful and wide-reaching duality. As a general rule, for every interesting definition, we obtain another interesting definition (the dual) by flipping all arrows. Moreover, for every theorem about the original definition, we obtain a theorem about the dual definition by flipping all arrows. This is studied in detail in the field of category theory [?, ?].

Submodels and Quotients

Consider a fixed model M of a fixed theory. We have the following duality between closed sets/submodels/images and congruences/quotients/kernels:

	submodels L of M	quotients Q of M
Constructed from	closed subsets of $\mathtt{term}^M$	congruence relations on $\mathtt{term}^M$
Characterized by	inclusion monomorphism $L \to M$	classification epimorphism $M \to Q$
Induced by	images of morphisms into M	kernels of morphisms out of M
Greatest model	M	$M/\Delta_M \cong M$
Least model	image of initial model	kernel of terminal model
Least model for $\Theta = \varnothing$	$Defin_M$	$M/\top_M$ (trivial model)

We can even unify closed subsets and congruence relations as follows:

Definition 10.40. An n-ary logical relation on M is a closed subset of M^n.

Now we have:

Theorem 10.41. *The closed subsets of M are the unary logical relations on M. The pre-congruences on M are the binary logical relations on M.*

Proof. The first result holds by definition.

For the second result, the key idea is that $M \times M$ is a Σ-model. The universe of a submodel of $M \times M$ is a subset of $\mathtt{term}^M \times \mathtt{term}^M$ (i.e., a binary relation on M) with an inclusion morphism into $M \times M$. Spelling out the morphism properties yields exactly the conditions on pre-congruences. □

Now a crucial observation is the following closure properties:

Theorem 10.42. *The closed subsets of M are closed under intersection.*

Proof. Exercise. □

Theorem 10.43. *The congruence relations of M are closed under intersection.*

Proof. Due to Thm. 10.42 and 10.41, the pre-congruences are closed under intersection. Moreover, we know that equivalence relations on $\mathtt{term}^M$ are just the submodels of $\top^M$ and thus by Thm. 10.42 closed under intersection. Therefore, the congruence relations are as well. □

Thus, the closed subsets and the congruence relations of M each form a closure system and thus a lattice.

Concretely, the lattice relation between submodels N, N' of M is given by the subset relation of their universes. Equivalently, $N \leq N'$ iff there is a monomorphism $N \hookrightarrow N'$. The greatest element is M itself. The least element is $Defin_M$. This is also the image of the unique morphism $I \to M$. The join of N and N' is the smallest submodel that is bigger than both N and N'; we can construct it by starting with $N \cup N'$ and iteratively adding elements until we obtain a closed subset.

The lattice relation between quotients Q, Q' of M is given by the subset relation between their congruence relations. However, we have to watch out for the following: the bigger the congruence relation, the smaller the quotient. We will write $Q \prec Q'$ if Q is smaller than Q', i.e., if the congruence of Q' is bigger than that of Q. Equivalently, $Q \prec Q'$ iff there is an epimorphism $Q' \to Q$. The greatest quotient is essentially M itself: M/Δ_M, which uses the least congruence relation Δ_M (which identifies nothing) and is isomorphic to M. The least quotient is $M/\top_M$, which uses the greatest congruence relation (which identifies everything). This is also the kernel of the unique morphism $M \to T$.

Note that contrary to the least quotient, the least submodel can have rich structure. For example, the least subring of the real numbers is the ring of integers.

Example 10.44.

- The mapping $k \mapsto (k\mathbb{Z}, +, 0, -, 1, \cdot)$ is a lattice morphism from $(\mathbb{N}, |^d, 1, 0)$ to the lattice of subrings of $\mathbb{Z}$.
  In particular, if $k|l$, then $l\mathbb{Z} \subseteq k\mathbb{Z}$ and the inclusion is a monomorphism.
  The least subring is $0\mathbb{Z}$; its universe is $\{0\}$.
  The greatest subring is $1\mathbb{Z}$; its universe is $\mathbb{Z}$.
- The mapping $k \mapsto (\mathbb{Z}_k, +_k, 0, -_k, 1, \cdot_k)$ is a lattice morphism from $(\mathbb{N}, |, 1, 0)$ to the lattice of quotients of $\mathbb{Z}$. (Technically, $\mathbb{Z}_k$ is not a quotient set of $\mathbb{Z}$. But it is isomorphic to the quotient $\mathbb{Z}/\equiv_k$.)
  In particular, if $k|l$, then $\mathbb{Z}_k \subseteq \mathbb{Z}_l$ and $modulus_k : \mathbb{Z}_l \to \mathbb{Z}_k$ is an epimorphism.
  The greatest quotient is $\mathbb{Z}/\equiv_0$, its universe is $\{\{z\} : z \in \mathbb{Z}\}$, which is isomorphic to $\mathbb{Z}_0 = \mathbb{Z}$.
  The least quotient is $\mathbb{Z}/\equiv_1$; its universe is $\{\mathbb{Z}\}$, which is isomorphic to $\mathbb{Z}_1 = \{0\}$.

Products and Coproducts

The dual concept of product models $M_1 \times M_2$ are the coproduct models $M_1 \sqcup M_2$.

For sets, the coproduct of sets is the disjoint union $D = \{1\} \times M_1 \cup \{2\} \times M_2$. Just like the product comes with two projection epimorphism $M_1 \times M_2 \to M_i$, the coproduct comes with two injection monomorphisms $M_i \to M_1 \sqcup M_2$.

However, it is not possible to define a coproduct in general for arbitrary signatures Σ. Moreover, if a coproduct of Σ-models M_i exists, its universe is usually not the disjoint union of the universes of the M_i.

10.9.5 Factorization of Morphisms

Theorem 10.45. *For every model morphism $\varphi : M \to N$, we have $M/(\ker \varphi) \cong N|(\operatorname{im} \varphi)$.*

Proof. The isomorphism maps

$$M/(\ker \varphi) \ \ni \ [m] \mapsto \varphi(m) \ \in \ N|(\operatorname{im} \varphi).$$

Its inverse maps

$$N|(\operatorname{im} \varphi) \ \ni \ n \mapsto \{m \in \mathtt{term}^M \mid \varphi(m) = n\} \ \in \ M/(\ker \varphi).$$

□

Theorem 10.46. *Every morphism $\varphi : M \to N$ is of the form $\varphi = e; m$ where e is epi and m is mono.*

Every morphism $\varphi : M \to N$ is uniquely of the form $\varphi = e; i; m$ where e is a classification, i is iso, and m is an inclusion.

Proof. The first statement follows easily from the second.

To prove the second statement, we pick the classification $e : M \to M/(\ker \varphi)$ and the inclusion $m : N|(\operatorname{im} \varphi) \to N$. The isomorphism i is then (uniquely) the one from Thm. 10.45. The uniqueness of m and e follows because no other classification would yield the same kernel as φ and no other inclusion the same image as φ. □

Part II

Higher-Order Logic

Chapter 11

Syntax

All basic concepts of first-order logic are preserved when going to higher-order logic (HOL, [?]): signatures, contexts, substitutions, terms, formulas. Two principle things change: types are added, and formulas become a special case of terms.

HOL without formulas yields simple type theory (STT, [?]), also called λ-calculus. We look at STT first and then obtain HOL from it.

11.1 Simple Type Theory

The **judgments** are:

$\vdash \Sigma$	Σ is a well-formed signature
$\vdash \sigma : \Sigma \to \Sigma'$	σ is a well-formed signature morphism from Σ to Σ'
$\vdash_\Sigma \Gamma\ \checkmark$	Γ is a well-formed Σ-context
$\vdash_\Sigma \gamma : \Gamma \to \Gamma'$	γ is a well-formed substitution from Γ to Γ' over Σ
$\vdash_\Sigma A : \mathtt{type}$	A is a well-formed type over Σ
$\Gamma \vdash_\Sigma t\ :\ A$	t is a well-formed term of (well-formed) type A over Σ and Γ

Before giving the inference system, let us look at a context-free **grammar** to get a better intuition:

$$\begin{array}{lcl}
\Sigma & ::= & \cdot \,|\, \Sigma,\ c : A \,|\, \Sigma,\ a : \mathtt{type} \\
\sigma & ::= & \cdot \,|\, \Sigma,\ c := t \,|\, \Sigma,\ a := A \\
\Gamma & ::= & \cdot \,|\, \Gamma,\ x : A \\
\gamma & ::= & \cdot \,|\, \gamma,\ x/t \\
A & ::= & a \,|\, A \to A \\
t & ::= & c \,|\, x \,|\, t\ t \,|\, \lambda x : A.t
\end{array}$$

Here $\cdot$ denotes the empty list. $\cdot$ is usually omitted in the judgments.
Intuitively, we have:

- Signatures are lists of typed constants ($c : A$) and base types (a).
- Morphisms are lists of expressions to be substituted for the constants ($c := t$) and base types ($a := A$).
- Contexts are lists of typed variables ($x : A$).
- Substitutions are lists of terms to be substituted for the variables (x/t).
- Types are base types (a) or function types ($A \to B$).
- Terms are constants (c), variables (x), function applications ($f\ t$), or λ-abstractions ($\lambda x : A.t$).

The inference system for these five judgments is not as simple as for FOL. In FOL, we could define first signatures, then contexts, then terms, then formulas. In STT, the definitions of **types** and **signatures** are mutally recursive as seen in Fig. 11.1 and 11.2.

$$\frac{\vdash \Sigma \quad a:\mathtt{type} \text{ in } \Sigma}{\vdash_\Sigma a:\mathtt{type}}\,\mathit{tpbase} \qquad \frac{\vdash_\Sigma A:\mathtt{type} \quad \vdash_\Sigma B:\mathtt{type}}{\vdash_\Sigma A \to B:\mathtt{type}}\,\mathit{tpfun}$$

Figure 11.1: Well-formed Types

$$\frac{}{\vdash \cdot}\,\mathit{sigempty} \qquad \frac{\vdash \Sigma \quad c \text{ not in } \Sigma \quad \vdash_\Sigma A:\mathtt{type}}{\vdash \Sigma,\ c:A}\,\mathit{sigcon} \qquad \frac{\vdash \Sigma \quad a \text{ not in } \Sigma}{\vdash \Sigma,\ a:\mathtt{type}}\,\mathit{sigtype}$$

Figure 11.2: Well-formed Signatures

Note that the rules are such that $\vdash_\Sigma A:\mathtt{type}$ implies $\vdash \Sigma$. This is a general principle: Whenever a judgment holds, all objects occuring in the judgment should be well-formed. Also note that types do not depend on contexts; this is clear from the grammar because no variables may occur in types.

Contexts can no longer be sets or lists of variables as for FOL – since the variables are typed, the definition of contexts depends on that of types. It is given in Fig. 11.3. The rules for **contexts** are very similar to those for signatures, only the case for base types is omitted, i.e., variables are always terms, never types.

$$\frac{\vdash \Sigma}{\vdash_\Sigma \cdot \checkmark}\,\mathit{conempty} \qquad \frac{\vdash_\Sigma \Gamma \checkmark \quad \vdash_\Sigma A:\mathtt{type}}{\vdash_\Sigma \Gamma,\ x:A \checkmark}\,\mathit{convar}$$

Figure 11.3: Well-formed Contexts

Note that the rules are such that $\vdash_\Sigma \Gamma \checkmark$ implies $\vdash \Sigma$. Also note that signatures may not declare the same name twice, but contexts may. If a context declares the same variable name twice, the later (more to the right) one takes precedence; we say it shadows the earlier declaration.

Substitutions are straightforward: The rules for **substitutions** have the same structure as those for contexts, see Fig. 11.4. A substitution from Γ to Γ' provides a Γ'-term for every variable in Γ. This is the same as for FOL except that the terms must be typed according to the types in Γ. Some expressions are underlined as a visual aid to make the bracketing structure apparent.

$$\frac{\vdash_\Sigma \Gamma' \checkmark}{\vdash_\Sigma \cdot : \cdot \to \Gamma'}\,\mathit{subsempty} \qquad \frac{\vdash_\Sigma \gamma:\Gamma \to \Gamma' \quad \Gamma' \vdash_\Sigma t\ :\ A}{\vdash_\Sigma \underline{\gamma,\ x/t} : \underline{\Gamma,\ x:A} \to \Gamma'}\,\mathit{subsvar}$$

Figure 11.4: Well-formed Substitutions

Note that a well-formed substitution from $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ to Γ' has the form $x_1/t_1, \ldots, x_n/t_n$ and satisfies $\Gamma' \vdash_\Sigma t_i\ :\ A_i$. Also note that the rules are such that $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ implies $\vdash_\Sigma \Gamma \checkmark$ (To see this, use the property (*) below.) and $\vdash_\Sigma \Gamma' \checkmark$.

While STT is more complicated conceptually than FOL – more judgments, mutual recursion between judgments – note that the rules of the inference system are actually very simple so far. Once we have an intuitive understanding of what we want to formalize, the rules are straightforward. The only rules that are a little bit more complicated are the ones for **terms**, see Fig. 11.5.

The difficulty is that we define two things at once: when a term is well-formed and what its type is. (Exactly the well-formed terms have types.) The first two rules are easy: Constants and variables are typed according to the signature and the context, respectively. The rule *termapp* formalizes function application: A function from A to B can be applied to an argument of type A yielding a result of type B. The rule *termlam* formalizes function formation: λ-abstraction over a term of type B with a free variable of type A yields a function from A to B.

Note how these rules are such that $\Gamma \vdash_\Sigma t\ :\ A$ implies $\vdash_\Sigma \Gamma \checkmark$ and $\vdash_\Sigma A:\mathtt{type}$ (*).

$$\frac{c : A \text{ in } \Sigma \quad \vdash_\Sigma \Gamma \checkmark}{\Gamma \vdash_\Sigma c \ : \ A}\,\textit{termcon} \qquad \frac{x : A \text{ in } \Gamma \text{ (rightmost } x \text{ if multiple)} \quad \vdash_\Sigma \Gamma \checkmark}{\Gamma \vdash_\Sigma x \ : \ A}\,\textit{termvar}$$

$$\frac{\Gamma \vdash_\Sigma f \ : \ A \to B \quad \Gamma \vdash_\Sigma t \ : \ A}{\Gamma \vdash_\Sigma f\ t \ : \ B}\,\textit{termapp} \qquad \frac{\Gamma,\ x : A \vdash_\Sigma t \ : \ B}{\Gamma \vdash_\Sigma \lambda x : A.t \ : \ A \to B}\,\textit{termlam}$$

Figure 11.5: Well-formed Terms

Notation 11.1. We use the following conventions to save brackets:

- $\to$ is right-associative, i.e., $A \to B \to C$ abbreviates $A \to (B \to C)$.
- iuxtaposition is left-associative, i.e., $f\ s\ t$ abbreviates $(f\ s)\ t$.

These abbreviation let us think of $A_1 \to \ldots \to A_n \to A$ as the type of n-ary functions with argument types $A_1, \ldots A_n$ and result type A. Similarly, if f is such a function and t_i has type A_i, we think of $f\ t_1\ \ldots\ t_n$ as the application of f to n arguments.

Finally, we can define substitution application and prove its intended property. It is notable that this is much easier for STT than FOL because there are only four cases.

Definition 11.2 (Substitution Application)**.** For a substitution $\gamma = x_1/t_1, \ldots, x_n/t_n$, and a term t using only the variables $x_1, \ldots, x_n$, we define $\overline{\gamma}(t)$ as follows:

- $\overline{\gamma}(c) = c$,
- $\overline{\gamma}(x_i) = t_i$,
- $\overline{\gamma}(f\ t) = \overline{\gamma}(f)\ \overline{\gamma}(t)$,
- $\overline{\gamma}(\lambda x : A.t) = \lambda x : A.\overline{\gamma^x}(t)$,

where γ^x abbreviates $\gamma, x/x$.

We also write $t[x/s]$ for the result of substituting x with s and all other variables with themselves.

Lemma 11.3 (Substitution Application)**.** *If* $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ *and* $\Gamma \vdash_\Sigma t \ : \ A$*, then* $\Gamma' \vdash_\Sigma \overline{\gamma}(t) \ : \ A$*.*

Proof. Exercise. □

Fig. 11.6 defines the rules for signature morphisms, which are very similar to those for substitutions: They map signature symbols to expressions, the preserved structure is the typing relation. Because variables canot occur in types, we did not need substitution in types. But signature symbols do occur in types (namely the base types); therefore, we need to define substitution in types as well and use it in the rule *morphcon*.

$$\frac{\vdash \Sigma'}{\vdash \cdot : \cdot \to \Sigma'}\,\textit{morphempty} \qquad \frac{\vdash \sigma : \Sigma \to \Sigma' \quad \cdot \vdash_{\Sigma'} t \ : \ \overline{\sigma}(A)}{\vdash \sigma,\ \underline{c := t} : \Sigma,\ \underline{c : A} \to \Sigma'}\,\textit{morphcon}$$

$$\frac{\vdash \sigma : \Sigma \to \Sigma' \quad \vdash_{\Sigma'} A : \texttt{type}}{\vdash \sigma,\ \underline{a := A} : \Sigma,\ \underline{a : \texttt{type}} \to \Sigma'}\,\textit{morphtype}$$

Figure 11.6: Well-formed Morphisms

Definition 11.4 (Morphism Application)**.** Assume a signature morphism $\vdash \sigma : \Sigma \to \Sigma'$. Then, for a Σ term or type, we define $\overline{\sigma}(-)$ as follows:

- $\overline{\sigma}(a) = A$ where $a := A$ in σ,
- $\overline{\sigma}(A \to B) = \overline{\sigma}(A) \to \overline{\sigma}(B)$,
- $\overline{\sigma}(c) = t$ where $c := t$ in σ,
- $\overline{\sigma}(x) = x$,
- $\overline{\sigma}(f\ t) = \overline{\sigma}(f)\ \overline{\sigma}(t)$,
- $\overline{\sigma}(\lambda x : A.t) = \lambda x : \overline{\sigma}(A).\overline{\sigma}(t)$.

Note that the context does not matter because all variables are mapped to themselves. More thoroughly, we could also define the application of signature morphism to contexts and substitutions and obtain a more general result below, but we omit that here and assume signature morphisms are only applied to closed expressions. That is a typical scenario.

Lemma 11.5 (Morphism Application)**.** *If* $\vdash \sigma : \Sigma \to \Sigma'$ *then: if* $\cdot \vdash_{\Sigma} t \ : \ A$*, then* $\cdot \vdash_{\Sigma'} \overline{\sigma}(t) \ : \ \overline{\sigma}(A)$*. If* $\vdash_{\Sigma} A : \mathtt{type}$*, then* $\vdash_{\Sigma} \overline{\sigma}(A) : \mathtt{type}$*.*

Proof. A straightforward induction on the derivations of well-formed expressions. □

11.2 Higher-Order Logic

Simple type theory is to higher-order logic as terms are to first-order logic, i.e., to obtain higher-order logic from simple type, we need to add the notion of formulas. One of the most appealing features of HOL is that formulas can be defined within simple type theory without changing the grammar or the inference system: All logical symbols of HOL can be defined as base types and constants of STT.

Let Σ_{HOL} be the pseudo-signature of STT containing the following declarations:

$o : \mathtt{type}$	base type for formulas/truth values
$\wedge : o \to o \to o$	constant for conjunction
$\vee : o \to o \to o$	constant for disjunction
$\Rightarrow : o \to o \to o$	constant for implication
$\neg : o \to o$	constant for negation
$\forall_A : (A \to o) \to o$	family of constants for universal quantification over type A
$\exists_A : (A \to o) \to o$	family of constants for existential quantification over type A
$\doteq_A : A \to A \to o$	family of constants for equality at type A

Note that we have to use $\Rightarrow$ for implication because $\to$ is already taken for function types. Some authors use $\supset$ for implication.

This is a pseudo-signature because it contains infinitely many constants. Namely, $\forall_A$, $\exists_A$, and $\doteq_A$ exist for every type that can be formed using all available base types. Apart from that, it is a well-formed STT-signature. We will pretend that this is a well-formed signature from now on, ignoring the fact that it is infinite. This is harmless because whenever we write $\Gamma \vdash_{\Sigma_{HOL}} t \ : \ A$, there are only finitely many constants that may occur in t. Thus, we can say that $\Gamma \vdash_{\Sigma_{HOL}} t \ : \ A$ abbreviates $\Gamma \vdash_{\Sigma} t \ : \ A$ for some finite fragment Σ of Σ_{HOL} that contains all instances of $\forall_B$, $\exists_B$, and $\doteq_B$ that occur in t. (They may not occur in Γ or A anyway.)

The type of equality depends on a parameter A: Equality takes two terms of the same but arbitrary type A and returns a formula.

The types of the universal and the existential quantifier look weird at first. They are an example of **higher-order abstract syntax**, which is the principle of using the λ-binder of the meta-language (here STT) to represent arbitrary binders of the object language (here HOL). Higher-order abstract syntax works because of the following equivalence

$$\Gamma, x : A \vdash_{\Sigma} F \ : \ o \qquad \text{iff} \qquad \Gamma \vdash_{\Sigma} \lambda x : A.F \ : \ A \to o.$$

It implies that terms of type $A \to o$ are in bijection to terms of type o with a free variable of type A. Because in $\forall x : A.F$, F must be a formula with a free variable x of type A, we can regard $\forall_A$ as a constant taking an argument of type $A \to o$. Thus, in HOL, quantifications are technically written as $\forall_A(\lambda x : A.F)$.

Notation 11.6. We introduce the following simplifications:

- If applied to two arguments, $\wedge$, $\vee$, $\Rightarrow$, and $\doteq_A$ are written infix. For example, $s \wedge t$ is the same as $\wedge\ s\ t$.
- $\forall x : A.F$ abbreviates $\forall_A(\lambda x : A.F)$.
- $\exists x : A.F$ abbreviates $\exists_A(\lambda x : A.F)$.
- $s \doteq t$ abbreviates $s \doteq_A t$. (This is possible because given s and t, we can always infer A.)

Then we are ready to define the syntax of HOL.

Definition 11.7. The collection $\mathbf{Sig}^{HOL}$ of HOL-**signatures** consists of the well-formed STT-signatures that do not declare a name already used in Σ_{HOL}.

Definition 11.8. The set $\mathbf{Sen}^{HOL}(\Sigma)$ of HOL-**sentences** over a HOL-signature Σ is the set

$$\{F \mid \vdash_{\Sigma_{HOL},\Sigma} F\ :\ o\}$$

of well-formed STT-terms of type o using constants from Σ_{HOL} or Σ.

This already defines the HOL-theories as pairs of a signature and a set of sentences.

Definition 11.9. The set of $\mathbf{Sig}^{HOL}$ **morphisms** from a HOL-signature Σ to a HOL-signature Σ' is the set of well-formed STT-signature morphisms between them.

Definition 11.10. For a HOL-signature morphism $\vdash \sigma : \Sigma \to \Sigma'$, the **sentence translation** $\mathbf{Sen}^{HOL}(\sigma) : \mathbf{Sen}^{HOL}(\Sigma) \to \mathbf{Sen}^{HOL}(\Sigma')$ **morphisms** is defined by morphism application: $\mathbf{Sen}^{HOL}(\sigma)(F) = \overline{\sigma}(F)$.

Example 11.11 (Natural Numbers). The theory of Peano arithmetic can be properly given as a finite HOL theory as follows:

- base type for the natural numbers: ι : `type`,
- constant for zero: $z : \iota$,
- constant for successor: $s : \iota \to \iota$,
- axiom for injectivity of successor: $\forall x : \iota \forall y : \iota.(s\ x \doteq s\ y \Rightarrow x \doteq y)$,
- axiom for zero as starting point: $\forall x : \iota.\neg(z \doteq s\ x)$,
- axiom for induction: $\forall P : \iota \to o.\big((P\ z \wedge \forall x : \iota.(P\ x \Rightarrow P\ (s\ x))) \Rightarrow \forall x : \iota.(P\ x)\big)$.

11.3 Minimal Sets of Logical Symbols

From FOL, it is well-known that we only need *false*, $\Rightarrow$, $\forall$, and $\doteq$ as primitive logical symbols. All others could be defined like this:

$$\begin{aligned}
\neg F &:= F \Rightarrow \mathit{false}\\
\mathit{true} &:= \neg\mathit{false} = \forall x.x \doteq x\\
F \vee G &:= \neg F \Rightarrow G\\
F \wedge G &:= \neg(\neg F \vee \neg G)\\
\exists x.F &:= \neg\forall x.\neg F
\end{aligned}$$

In HOL, we can go even further: We can still define all logical symbols if only $\doteq$ is primitive. We can define as follows:

$$\begin{array}{ll}
\mathit{true} & := (\lambda x : o.x) \doteq_{o\to o} (\lambda x : o.x) \\
\forall_A & := \lambda f : A \to o.(f \doteq_{A\to o} \lambda x : A.\mathit{true}) \\
\mathit{false} & := \forall x : o.(x \doteq_o \mathit{true}) \\
\neg & := \lambda x : o.(x \doteq_o \mathit{false}) \\
\wedge & := \lambda x : o.\lambda y : o.\forall f : o \to o \to o.((f\ x\ y) \doteq_o (f\ \mathit{true}\ \mathit{true})) \\
\Rightarrow & := \lambda x : o.\lambda y : o.((x \wedge y) \doteq_o x) \\
\vee & := \lambda x : o.\lambda y : o.\forall c : o.(((x \Rightarrow c) \wedge (y \Rightarrow c)) \Rightarrow c) \\
\exists_A & := \lambda f : A \to o.\forall c : o.((\forall x : A.(f\ x \Rightarrow c)) \Rightarrow c)
\end{array}$$

Thus, STT with a base type o and a family of constants $\doteq_A$ is already enough to obtain higher-order logic.

Chapter 12

Model-Theoretic Semantics

12.1 The Naive Semantics

The semantics of HOL is defined similarly to the semantics of FOL: We fix interpretations of the logical symbols, use models to interpret the constants, and use assignments to interpret the variables.

However, because the definitions of types and signatures are mutually recursive, we cannot define models, assignments, and the interpretation function $[\![-]\!]^{I,\alpha}$ separately anymore. Instead, we use a mutual induction.

We start with an overview. For a Σ-model I and an assignment α for a Σ-context Γ, the relation between syntax and semantics is as follows:

Expression	Syntax	Semantics
type A	$\vdash_\Sigma A : \mathtt{type}$	$[\![A]\!]^I \in \mathcal{SET}$
term t	$\Gamma \vdash_\Sigma t \ : \ A$	$[\![\Gamma \vert t]\!]^{I,\alpha} \in [\![A]\!]^I$
formula F	$\Gamma \vdash_\Sigma F \ : \ o$	$[\![\Gamma \vert F]\!]^{I,\alpha} \in [\![o]\!]^I = \{0,1\}$

Note that because variables do not occur in types, the semantics of types only depends on the model and not on the assignment.

Definition 12.1 (Models, Assignments, Semantics of STT). A *model* I of an STT-signature Σ is defined as follows:

- If $\Sigma = \cdot$, then I is the empty tuple.
- If $\Sigma = \Sigma_0, a : \mathtt{type}$, then I is a pair (I_0, a^I) where I_0 is a model of Σ_0 and a^I is a set.
- If $\Sigma = \Sigma_0, c : A$, then I is a pair (I_0, c^I) where I_0 is a model of Σ_0 and $c^I \in [\![A]\!]^I$.

An assignment α from a Σ-context Γ into a Σ-model I is defined as follows:

- If $\Gamma = \cdot$, then α is the empty function.
- If $\Gamma = \Gamma_0, x : A$, then α is a function $\alpha_0 \cup \{x/u\}$ where α_0 is an assignment for Γ_0 and $u \in [\![A]\!]^I$.

For a signature Σ, a context Γ, a model I, and an assignment α, the semantics of types and terms in context Γ is defined as follows:

- $[\![a]\!]^I = a^I$,
- $[\![A \to B]\!]^I = ([\![B]\!]^I)^{[\![A]\!]^I}$,
- $[\![\Gamma|c]\!]^{I,\alpha} = c^I$,
- $[\![\Gamma|x]\!]^{I,\alpha} = \alpha(x)$,
- $[\![\Gamma|f\ t]\!]^{I,\alpha} = [\![\Gamma|f]\!]^{I,\alpha}([\![\Gamma|t]\!]^{I,\alpha})$,
- $[\![\Gamma|\lambda x : A.t]\!]^{I,\alpha} = \{(u, [\![\Gamma, x : A|t]\!]^{I,\alpha\cup\{x/u\}}) \ : \ u \in [\![A]\!]^I\}$.

Then, to define the semantics of HOL, we only have to fix interpretations for the symbols in Σ_{HOL}. In Def. 12.2, we will only fix the interpretations of o and $\doteq_A$; the interpretation of the remaining symbols is induced by the abbreviations from Sect. 11.3. Then we proof in Lem. 12.3 that these abbreviations do indeed yield the intended interpretations for them.

Definition 12.2 (Semantics of HOL)**.** We abbreviate $\mathbb{B} = \{0,1\}$. A HOL-model of the HOL-signature Σ is an STT-model of Σ such that (i) all base types are interpreted as non-empty sets, and (ii) the symbols of Σ_{HOL} are interpreted as follows:

- $o^I = \mathbb{B}$,
- $\doteq_A^I \in \left(\mathbb{B}^{[\![A]\!]^I}\right)^{[\![A]\!]^I}$ such that for all $a, b \in [\![A]\!]^I$ $\quad \doteq_A^I (a)(b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$

For $I \in \mathbf{Mod}^{HOL}(\Sigma)$ and $F \in \mathbf{Sen}^{HOL}(\Sigma)$, we put

$$I \models F \qquad \text{iff} \qquad [\![F]\!]^I = 1.$$

As for FOL, this induces the model-theoretic consequence.

Lemma 12.3. *Using the abbreviations from Sect. 11.3, we have for all $u, v \in \mathbb{B}$ and all $\varphi : [\![A]\!]^I \to \mathbb{B}$:*

- $\mathit{true}^I = 1$,
- $\mathit{false}^I = 0$,
- $\neg^I(u) = 1 - u$,
- $\wedge^I(u)(v) = \min\{u, v\}$,
- $\vee^I(u)(v) = \max\{u, v\}$,
- $\Rightarrow^I (u)(v) = \begin{cases} 1 & \text{if } u \leq v \\ 0 & \text{otherwise,} \end{cases}$
- $(\forall_A)^I(\varphi) = \min_{a \in [\![A]\!]^I} \varphi(a) = \begin{cases} 1 & \text{if } \varphi(a) = 1 \text{ for all } a \in [\![A]\!]^I \\ 0 & \text{otherwise,} \end{cases}$
- $(\exists_A)^I(\varphi) = \max_{a \in [\![A]\!]^I} \varphi(a) = \begin{cases} 1 & \text{if } \varphi(a) = 1 \text{ for some } a \in [\![A]\!]^I \\ 0 & \text{otherwise.} \end{cases}$

Proof. We prove the case for $\neg$ as example and leave the rest as an exercise.
Expanding the definition of $\neg$:

$$\neg^I(u) = [\![\cdot|\neg]\!]^I(u) = [\![\cdot|\lambda x : o.x \doteq_o \mathit{false}]\!]^I(u)$$

Applying the semantics of λ-abstraction:

$$= \left\{ \left(v, [\![x : o|x \doteq_o \mathit{false}]\!]^{I,x/v}\right) \; : \; v \in [\![o]\!]^I \right\} (u)$$

Applying the function $\left\{ \ldots \right\}$ to the argument u:

$$= [\![x : o|x \doteq_o \mathit{false}]\!]^{I,x/u}$$

Applying the semantics of application twice:

$$= [\![x : o| \doteq_o]\!]^{I,x/u}\left([\![x : o|x]\!]^{I,x/u}\right)\left([\![x : o|\mathit{false}]\!]^{I,x/u}\right)$$

Applying the semantics of variables (to x) and constants (to $\doteq_o$ and *false*):

$$= \doteq_o^I (u, \mathit{false}^I)$$

Applying the semantics of $\doteq_o$ and assuming we have proved $\mathit{false}^I = 0$ already:

$$= \left\{ \begin{matrix} 1 & \text{if } u = \mathit{false}^I \\ 0 & \text{otherwise} \end{matrix} \right\} = 1 - u$$

□

Remark 12.4. The restriction (i) in Def. 12.2 implies that all types are interpreted as non-empty sets.

Remark 12.5. Just like for the syntax, we find that the semantics of HOL is conceptually more complicated as for FOL – mutual recursion for models and semantics – but technically simpler than FOL.

Chapter 13

Proof-Theoretic Semantics

As for FOL, for the proof theoretic semantics of HOL, we add a judgment of provability:

$$\Gamma;\Delta \vdash_\Sigma F.$$

The proof rules that define this judgment are given in Fig. 13.1, where the less relevant parts are kept in gray. We use the abbreviations from Sect. 11.3 and only give rules for $\doteq_A$; the usual rules for the other symbols are derived in Lem. 13.2. Note that the rules are such that $\Gamma;\Delta \vdash_\Sigma F$ implies $\Gamma \vdash_\Sigma F \ : \ o$.

$$\frac{\Gamma \vdash_\Sigma t \ : \ A}{\Gamma;\Delta \vdash_\Sigma t \doteq_A t}\,eqrefl \qquad \frac{\Gamma;\Delta \vdash_\Sigma s \doteq_A t}{\Gamma;\Delta \vdash_\Sigma t \doteq_A s}\,eqsym$$

$$\frac{\Gamma;\Delta \vdash_\Sigma f \doteq_{A\to B} f' \quad \Gamma;\Delta \vdash_\Sigma t \doteq_A t'}{\Gamma;\Delta \vdash_\Sigma (f\ t) \doteq_B (f'\ t')}\,eqapp \qquad \frac{\Gamma,\ x : A;\Delta \vdash_\Sigma t \doteq_B t'}{\Gamma;\Delta \vdash_\Sigma (\lambda x : A.t) \doteq_{A\to B} (\lambda x : A.t')}\,eqlam$$

$$\frac{\Gamma,\ x : A \vdash_\Sigma t \ : \ B \quad \Gamma \vdash_\Sigma s \ : \ A}{\Gamma;\Delta \vdash_\Sigma (\lambda x : A.t)\ s \doteq_B t[x/s]}\,beta \qquad \frac{\Gamma \vdash_\Sigma f \ : \ A \to B \quad x \text{ not in } \Gamma}{\Gamma;\Delta \vdash_\Sigma f \doteq_{A\to B} \lambda x : A.(f\ x)}\,eta$$

$$\frac{\Gamma;\Delta, F \vdash_\Sigma G \quad \Gamma;\Delta, G \vdash_\Sigma F}{\Gamma;\Delta \vdash_\Sigma F \doteq_o G}\doteq_o I \qquad \frac{\Gamma;\Delta \vdash_\Sigma F \doteq_o G \quad \Gamma;\Delta \vdash_\Sigma F}{\Gamma;\Delta \vdash_\Sigma G}\doteq_o E$$

$$\frac{F \in \Delta \quad \Gamma \vdash_\Sigma F \ : \ o}{\Gamma;\Delta \vdash_\Sigma F}\,axiom \qquad \frac{\vdash_\Sigma \Gamma\ \checkmark}{\Gamma;\Delta \vdash_\Sigma \forall x : o.(x \doteq_o true \lor x \doteq_o false)}\,tnd$$

$$\frac{\Gamma, x : A;\Delta \vdash_\Sigma F \quad x \notin \Gamma, \Delta, F}{\Gamma;\Delta \vdash_\Sigma F}\,nonempty \qquad \frac{\Gamma;\Delta \vdash_\Sigma F \quad \Gamma;\Delta, F \vdash_\Sigma G}{\Gamma;\Delta \vdash_\Sigma G}\,cut$$

Figure 13.1: Proof Rules

The intuitions behind the rules are as follows.

- *eqrefl*, *eqsym*: These rules are reflexivity and symmetry and make equality an equivalence relation on terms. (Transitivity is derivable – exercise.)
- *eqapp*, *eqlam*: These rules make equality a congruence relation. *eqapp* says that applying equal functions to equal arguments yields equal results. *eqlam* says that λ-abstraction over equal terms yields equal results.

- *beta*, *eta*: These rules axiomatize the meaning of functions. *beta* says that function application means to substitute the argument for the λ-abstracted variable.
- $\doteq_o I$, $\doteq_o E$: These rules express the intuition that equality between formulas is logical equivalence, i.e., $F \doteq_o G$ is the same as $(F \Rightarrow G) \wedge (G \Rightarrow F)$.
- *tnd*: tertium non datur as for FOL.
- *axiom*: axiom rule as for FOL.
- *nonempty*: Similar to FOL, this rule makes sure that all types are non-empty.
- *cut*: cut rule as for FOL.

As for FOL, we define proof-theoretic consequence.

Definition 13.1. We say that $F \in \mathbf{Sen}^{HOL}(\Sigma)$ is a proof-theoretical *consequence* of the set of assumptions $\Delta \subseteq \mathbf{Sen}^{HOL}(\Sigma)$ if there is a derivation of $\varnothing; \Delta \vdash F$ in the calculus above.

Lemma 13.2. *Using the abbreviations from Sect. 11.3, the natural deduction rules for introduction and elimination of truth, falsity, conjunction, disjunction, implication, and negation as given for FOL are derivable in HOL. Similarly, the following rules for universal and existential quantification are derivable in HOL:*

$$\frac{\Gamma; \Delta \vdash_\Sigma \forall x : A.F}{\Gamma, x : A; \Delta \vdash_\Sigma F \qquad x \notin \Gamma, \Delta}\forall_A I \qquad\qquad \frac{\Gamma; \Delta \vdash_\Sigma F[x/t]}{\Gamma; \Delta \vdash_\Sigma \forall x : A.F \qquad \Gamma \vdash_\Sigma t \;:\; A}\forall_A E$$

$$\frac{\Gamma; \Delta \vdash_\Sigma \exists x : A.F}{\Gamma; \Delta \vdash_\Sigma F[x/t] \qquad \Gamma \vdash_\Sigma t \;:\; A}\exists_A I \qquad\qquad \frac{\Gamma; \Delta \vdash_\Sigma C}{\Gamma; \Delta \vdash_\Sigma \exists x : A.F \qquad \Gamma, x : A; \Delta, F \vdash_\Sigma C \qquad x \notin \Gamma, \Delta, C}\exists_A E$$

Proof. We show the case of negation introduction as an example and leave the rest as an exercise.
Using the definition $\neg := \lambda x : o.(x \doteq_o \mathit{false})$, the rule for negation introduction is

$$\frac{\Gamma; \Delta \vdash_\Sigma (\lambda x : o.(x \doteq_o \mathit{false}))\ F}{\Gamma; \Delta, F \vdash_\Sigma \mathit{false}}\neg I$$

To show that it is a derivable rule means to show that there is a derivation of $\Gamma; \Delta \vdash_\Sigma (\lambda x : o.(x \doteq_o \mathit{false}))\ F$ using an assumption $\Gamma; \Delta, F \vdash_\Sigma \mathit{false}$. Such a derivation is this where we assume that we have derived the rule *falseE* already:

$$\frac{\Gamma; \Delta \vdash_\Sigma (\lambda x : o.(x \doteq_o \mathit{false}))\ F}{\dfrac{\Gamma; \Delta \vdash_\Sigma (F \doteq_o \mathit{false}) \doteq_o ((\lambda x : o.(x \doteq_o \mathit{false}))\ F)}{\dfrac{\Gamma; \Delta \vdash_\Sigma ((\lambda x : o.(x \doteq_o \mathit{false}))\ F) \doteq_o (F \doteq_o \mathit{false})}{}\mathit{beta}}\mathit{eqsym} \qquad \dfrac{\Gamma; \Delta \vdash_\Sigma F \doteq_o \mathit{false}}{\Gamma; \Delta, F \vdash_\Sigma \mathit{false} \qquad \dfrac{\Gamma; \Delta, \mathit{false} \vdash_\Sigma F}{}\mathit{falseE}}\doteq_o I}\doteq_o E$$

□

Chapter 14

The Relation between Proof and Model Theory

Theorem 14.1. *HOL is sound.*

Proof. Exercise. □

We do not have completeness for HOL: Not all sentences that are true in all models are provable. There are two ways to interpret that:

- From a model-theoretical perspective, the model theory is fine but more axioms and proof rules are needed to make all theorems provable. But the model theory of HOL is sufficient for the natural numbers and large parts of mathematics. Due to Gödel's result, the set of theorems is not recursively enumerable, i.e., no complete calculus (with a decidable set of rules) exists.
- From a proof-theoretical perspective, the proof theory is fine but more models are needed to make less sentences theorems. Completeness holds when using more general models. But these models are not intuitive anymore: Either the interpretation of function types is unnatural (when using Henkin-models) or the interpretation of base types is unnatural (when using cartesian closed categories).

Part III

Foundations of Mathematics

Chapter 15

General Ideas

15.1 Background

History The discovery of paradoxa, i.e., contradictions, in what is called **naive set theory** in retrospect caused the Grundlagenkrise around 1900. Naive set theory was the implicitly assumed foundation of mathematics at the time, Cantor's Grundlagen ([?]) from 1883 being the most influential contribution. The best known paradoxon was found by Russell in 1901 ([?]). Peano had noticed a similar one in 1897.

Roughly, Russell's paradoxon arises from unlimited set comprehension. That leads to a contradiction because it permits to form the set of all sets that do not contain themselves. Intuitively, a contradiction in a logic means that something is both true and not true. That typically makes everything true, by which truth becomes vacuous. Since mathematics is a strictly hierarchical science with every new concept resting on the preceding ones, a contradiction in mathematics, unless it can be remedied somehow, is tantamount to total destruction. Therefore, the freeness from contradictions, called **consistency**, is crucial for a foundation.

In response to this, mathematicians have developed several — sometimes alternative, sometimes complementary — foundations that can replace naive set theory. This happened over several decades as an evolutionary creative process. But it did not culminate in a commonly accepted solution. Rather, it led to profound and sometimes fierce debates on what mathematics is. The personal quarrel between Hilbert and Brouwer, which was partially fuelled by these debates, is an almost tragic example. From this evolution emerged two major classes of foundations: axiomatic set theory and type theory. (The clear separation between set and type theory is partially drawn here for instructive purposes: The historical and mathematical boundaries are not as sharp.)

The basic idea of **axiomatic set theory** is that there is a universe of sets, and any mathematical object ever introduced is a set. The sets are related via the binary relations of equality and membership. For example $m \in M$ is used to say that the set m is a member of the set M. Depending on context, M is regarded as a property of m or as a structuring concept.

To talk about sets, equality, and membership, propositions are used. The basic propositions are of the form $m = m'$ and $m \in M$. Composed propositions are built up from the basic ones. Typically, (at least) FOL is used as the language of composed propositions: FOL uses propositions such as $F \wedge G$ and $\forall x.F(x)$ denoting "F and G are true" and "for all (sets) x, F is true about x".

Type theory mainly differs from set theory in that it employs a stratification of the mathematical universe. In the simplest type theories, the basic concepts are *term* and type. Intuitively, terms represent mathematical objects, and types represent properties and structuring concepts. And for a term m and a type M, the propositions (in the context of type theory often called judgments) $m = m$, $M = M$, and $m : M$ are used, where the first two state equality and the third is used to say that m has type M. Thus, the cases $m : m$ and $M : M$ leading to the paradoxon of naive set theory are excluded by construction. Often a term may only have one type, which is in contrast to set theory, where a set may be a member of arbitrary many other sets. In that case there is a characteristic contrast between the universe of sets in set theory and the typed terms of type theory.

Both set theory and type theory have led to numerous specific foundations of mathematics. Zermelo-Fraenkel set theory, based on [?, ?], is most commonly in use today. Other variants are von Neumann-Bernays-Gödel set theory, based on [?, ?, ?], which is important for category theory, and Tarski-Groethendieck set theory, based on [?, ?]. The first type theory was Russells's ramified theory of types ([?]). And in their Principia ([?]), Whitehead and Russell gave one of the most influential foundations of mathematics. Church's simple theory of types, also called higher-order logic, ([?]) is the most-used type theory today. Important other type theories are typically organized in the lambda cube ([?]) and include dependent type theory ([?, ?]), System F ([?, ?]), and the calculus of constructions ([?]). Most of these foundations have further variants, such as Zermelo-Fraenkel set theory with or without the Axiom of Choice or type theory with or without product types.

Philosophy of Mathematics Besides the set/type theory distinction, there are other dimensions along which foundations can be distinguished. We will only briefly mention the question of the philosophical nature of mathematics. In **platonism** (going back to Plato, with various defenders in the 20th century, e.g., K. Gödel), formal mathematical objects are only devised as representations of abstract platonic objects to assist in reasoning. And within the non-platonistic view, four important schools can be singled out. In **formalism** (main proponent D. Hilbert, see, e.g., [?]), the formal mathematical objects themselves are the objects of interest. Thus, mathematical reasoning can be reduced to purely mechanical procedures. In **logicism** (main proponent G. Frege, see [?]), all of mathematics is reduced to logic, i.e., all axioms are logical truths without mathematical intuition. In **intuitionism** (main proponent L. Brouwer, see [?]), truth depends on a mathematician's experience of it. Thus, mathematical proofs are only supplements of mental constructions. Intuitionism rejects, e.g., an argument that derives "F" from "not not F" because the absence of the truth of "not F" does not yield the truth of "F". And **predicativism** (main proponents H. Weyl, S. Feferman, see [?, ?]) emphasizes the need that a mathematical definition may not depend on the defined object itself. Predicativism rejects, e.g., the definition of a closure of a set as the intersection of all supersets with a certain property because the closure is among the intersected sets.

The correlation of these philosophical views to each other and to the set/type theory distinction is complex. Most mathematicians tend to think platonistically, and platonists tend to favor set theory over type theory. Computer-supported work tends to be formalistic. But formalism is not strongly correlated with either set or type theory. Type theories are usually predicative and often intuitionistic whereas set theories are usually neither. Similarly, researchers favoring type theory tend to favor intuitionism and predicativism. The importance of logicism has faded in general. A good overview is given in [?].

Frege	1879	first mordern logic
Cantor	1883	(naive) set theory
Peano	1889	axioms for natural numbers
Russell	1901	paradoxon in naive (= informal) set theory, Grundlagenkrise
Brouwer	1900s	intuitionism
Russell, Whitehead	1900s	first type theory (Principia)
Zermelo, Fraenkel	about 1910s	first axiomatic set theory
Hilbert	1920s	Hilbert's program (reduction of mathematics to formal methods)
Gödel	1930	incompleteness results, impossibility of Hilbert's program
Gödel, Gentzen	1930s	first-order logic
Church	1930s	lambda calculus, higher-order logic
de Bruijn	1970s	first formalized (= computerized) foundation (Automath)
various	1970s, 1980s	advanced type theories (dependent types, calculus of constructions, LF)
various	since 1980s	advanced formalized foundations (Mizar, HOL, Coq)

15.2 Foundations as Informal Concepts

Foundations and Logics A foundation of mathematics is a formal language *for mathematics itself.*

This is different from a logic: Logics are formal languages developed *within mathematics.* In the grand hierarchy of mathematics, the foundation is the very first thing we introduce. Then, using the foundation, we develop all objects of mathematics interest, such as sets, functions, numbers, algebraic structures, and also logics.

On the other hand, foundations and logics are very similar: Both are formal languages used to describe mathematical objects. In fact, foundations usually arise by fixing a logic and one logical theory in it. For example, axiomatic set theory from Ex. 1.32 is a foundation that arises by fixing a theory of the logic FOLEQ.

To disentangle the relationship between logics and foundations, let us look at the dependency between them. Logics consist of three parts: syntax, proof theory, and model theory. We have seen that the syntax and the proof theory

of a logic can be described using only very basic mathematical tools: grammars and inference systems. And these two tools do not depend on anything else in mathematics – they are just producing and manipulating strings. [1]
The situation is different for the model theory: Models interpret the syntax in mathematics. Since every model may make use of any mathematical object, the model theory of a logic may depend on everything else in mathematics. So far we have been a bit vague about what constitutes "mathematics" – we have simply assumed an ambient language in which we can express whatever mathematical intuitions we have. We can now make this more precise: The ambient language is the foundation, and a model is an interpretation of the syntax in the foundation.

Thus, *proof-theoretic logics*, i.e., languages which have a syntax and proof theory but no model theory, can be given without reference to other areas of mathematics. Now we can say: A *foundation* arises by fixing a proof-theoretic logic and a theory in it. We speak of the *foundational logic* and the *foundational theory*. We cannot give a model theoretic semantics for the foundation itself – it is inherent in the nature of the foundation that there is no other language in which we could formally interpret it.

The Definitional Method We arrive at the following blueprint to build mathematics:

1. We define grammars and inference systems.[2]
2. We use them to define proof-theoretic logics.
3. We fix one proof-theoretical logic and one theory in it as the foundation.
4. We develop mathematics within this theory. In particular, we give model theories for other logics.

Developing mathematics in a foundation means that every mathematical object is formally defined as an expression in the foundation. Mathematical theorems are the provable sentences of that theory. For example, if ST is the FOLEQ-theory for set theory, and we fix ST as the foundation, then every mathematical object is a FOLEQ-term over ST, and every mathematical sentence is a FOLEQ-sentence.
This is called the *definitional method*: Every new object is introduced by defining it in terms of previously defined objects. The only primitives objects are those symbols introduced by the foundation. Similarly, every new theorem is proved from previously established theorems. The only primitive theorems are the axioms introduced by the foundation.

Definition Principles Interestingly, the above blueprint is not quite correct. It turns out that the definitional method is rather weak: Often less objects can be defined than we want to define. This could, in principle, be remedied by changing the logic accordingly. Arguably, it should be remedied this way. But this is not what scholars have preferred in the past.
Instead, besides fixing a logic and a theory in it, they usually introduce a third entity: *definition principles* for *conservative extensions*. A definition principle adds some new primitive symbol and some new axioms to the foundation in such a way that the new foundation is a conservative extension of the original one. Here "conservative" means the semantics is retained in the following sense: The extended foundations agrees with the old foundation regarding which sentences of the old foundation are theorems and which are not.
The most important example of a definition principle is *description*. Assume our foundations has a quantifier $\exists^!$ for unique existence, and assume we have proved $\exists^! x.F(x)$. Then the description principle, lets us add a new principle symbol c together with an axiom $F(c)$. The intuition is that we introduce c as a name for the uniquely existing object.

Leaps of Faith The appeal of the definitional method is that the problem of inconsistency is reduced to the choice of the foundation. If the axiom of the foundation are chosen consistently, then the definitional method can never introduce inconsistencies. Thus, the foundation provides a solid, robust footing to start building mathematics.
But how can we be sure that the foundation is consistent. The best way to prove consistency of a theory is to give a model. But we cannot give a model theory for the foundation itself. Even if we could somehow manage to prove consistency — e.g., by giving a model the foundation within itself — this would not help: Using the foundation to prove its own consistency is just circular reasoning — of course, an inconsistent foundation can prove anything

[1] Readers already familiar with LF from Part V should note that this is exactly what LF provides. Thus, if we take LF for granted, we can define all the other grammars and inference systems within LF.
[2] Readers already familiar with LF from Part V may read this as: We introduce LF.

including its own consistency. In fact, Gödel's theorem goes even further: It basically says that the only foundations that can prove their own consistency are the inconsistent ones.

Therefore, we have to trust the consistency of the foundation. Understandably, this has led to a substantial amount of disagreement between scholars arguing which foundation to use. Usually, we try to use the weakest possible foundation, the one that requires the least trust. A foundation should only formalize ideas that are immediately obvious and somehow justified by our intuition and reality.

Due to this disagreement about which leaps of faith to take, scholars have not agreed on a joint foundation and probably never will. Instead, multiple foundations exist, and each one of them is used to develop mathematics. Moreover, if we have multiple foundations, we can use of them to give model theories for each other. This amounts to translating between foundations. While this does not prove consistency of any one of them (After all, they might all be inconsistent.), this kind of cross-verification gives additional trust in them.

The Axiomatic Method The definitional method fixes a logic and a theory as the foundation. The axiomatic method on the contrary uses multiple logics and theories at the same time. Where the definitional method takes care to define every new object in terms of the previously defined ones, the axiomatic method simply switches to other theories in which new primitive symbols and axioms are added.

For example, in set theory, the definitional method introduces the natural numbers (using $0 := \varnothing$, $1 := \{\varnothing\}$, etc.), then the integers (using pairs (a, b) of natural numbers quotiented by $(a, b) \sim (a', b')$ iff $a + b' = b + a'$), then the rational numbers (similarly, using pairs (a, b) of integers), and finally the real numbers (using Cauchy sequences of rational numbers or Dedekind cuts of the set of rational numbers). The axiomatic method simply introduces the real numbers using a theory similar to the one for the natural numbers in Ex. 1.35.

There are two key differences between the two methods. Firstly, the axiomatic method is much simpler: It does not bother us with the details of how the numbers are defined. But it pays a price: Consistency is not guaranteed at all because axioms are introduced freely. Secondly, the definitional method constructs the real numbers as objects in the foundation; so we know exactly what they are. But it pays a price, too: The construction depends on the foundation and only works in this foundation.

The axiomatic method can be seen as a complement to the definitional method that abstracts from the construction and only works with the final result. Axiomatically, we pick a logic L and a theory T and do some work in it, e.g., by proving $T \vdash^L F$. Definitionally, we fix a foundation $\mathcal{F}$ and give a sound model theory M for L and a model for T in it. Because of soundness, we can port the axiomatic result to the definitional realm and conclude that $M \models^{\mathcal{F}} F$. Most importantly, the axiomatic result can be reused in any foundation, in which a model M can be found. If T is the theory of the real numbers, it does not matter if M is constructed in set theory using Cauchy sequences or in any other way in any other foundation. Scholars use this principle implicitly all the time and thus gain a certain degree of independence from the foundation.

Most results in mathematics are separated from the foundation by several layers of axiomatic theories. It is understood that a result applies to any foundation in which these axiomatic theories can be modeled. Since most mathematically interesting axiomatic theories can be modeled in each foundation, (If not, it is arguably a bad foundation.), it often does not matter which foundation a result is based on.

15.3 An Abstract Definition of a Foundation

We can give a formal definition of a foundation in the following way:

Definition 15.1 (Foundation)**.** A *foundation* consists of a proof theoretical logic L and a theory $\mathcal{F}$ of L.

This definition does not account for definition principles yet. Let us first define:

Definition 15.2. In a logic $L = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash)$, a theory extension $(\Sigma, \Theta) \hookrightarrow (\Sigma', \Theta')$ is *conservative* if

$$\Theta \vdash_{\Sigma} F \qquad \text{iff} \qquad \Theta' \vdash_{\Sigma'} F \qquad \text{for all } F \in \mathbf{Sen}(\Sigma)$$

Thus, an extension is conservative if the provability of the old sentences (i.e., the ones from Σ) is not affected when switching to (Σ', Θ'). That is exactly when enriching our foundation with a definition principle.

Thus, we can say: A definition principle is a sufficient criterion for when a theory extension in L is conservative. Then, appealing to a definition principle means to switch from a theory T to a conservative extension of T. In particular, developing mathematics in $\mathcal{F}$ is about building a tree of conservative extensions with $\mathcal{F}$ at the root.

The axiomatic approach is similar except that it studies all extensions, including the non-conservative ones. Therefore, the axiomatic approach works with a collection of trees of conservative extensions. Whenever a non-conservative extension is used, a new tree must be started.

Theory morphisms in L permit moving theorems between these trees. Here we can make use of conservative extensions as follows: We can choose the domain as small as possible (i.e., the root of one these trees); that is good because there are less things to prove. And we can make the codomain as large as possible (i.e., some node high up in a tree of conservative extensions); that is good because there are more things available to do the proof.

Chapter 16

First-Order Set Theory

Almost all variants of set theory use first-order logic but they differ in the fixed signature and axioms. All of them can be used to develop most of mathematics, but there are subtle differences in exactly what can be developed and how conveniently.

16.1 Foundational Logic and Theory

The underlying of these foundations is FOLEQ as defined in Part I. The details of the accounts vary. For example, sometimes only some connectives are primitive and the others are defined. Or different calculi are used to define the same collection of theorems. None of these differences are essential. In fact, most authors are mathematicians and not logicians and therefore assume FOLEQ implicitly without bothering to define it formally.

There is only one substantial difference that may exist: the difference between classical and intuitionistic FOLEQ. The former uses the axiom $F \vee \neg F$ of excluded middle, the former does not. Almost all mathematicians use classical FOLEQ, but it is generally worth keeping track of which results require classical reasoning. Often the use of excluded middle can be avoided or worked around.

The characteristic feature of set theories is that their signature has a single binary predicate symbol $\in$ and no function symbols. We will give a Zermelo-Fraenkel-style variant of set theory.

Definition 16.1. ZF is the FOLEQ theory consisting of no function symbols, a binary predicate symbol $\in$ (written infix), and the following axioms

- extensionality ("Two sets are equal if they have the same elements.")

$$\forall X\ \forall Y\ (X \doteq Y \leftrightarrow \forall z\ (z \in X \leftrightarrow z \in Y))$$

- empty set ("There is a set that we can call $\varnothing$.")

$$\exists E\ \forall z\ \neg z \in E$$

- pairing ("For all X, Y, there is a set that we may call $\{X, Y\}$.")

$$\forall X\ \forall Y\ \exists P\ \forall z\ (z \in P \leftrightarrow (z \doteq X \vee z \doteq Y))$$

- union ("For all X, there is a set that we may call $\bigcup_{x \in X} x$.")

$$\forall X\ \exists U\ \forall z\ (z \in U \leftrightarrow \exists x\ (x \in X \wedge z \in x))$$

- power set ("For all X, there is a set that we may call $\mathcal{P}(X)$.")

$$\forall X\ \exists P\ \forall z\ (z \in P \leftrightarrow \forall x\ (x \in z \rightarrow x \in X))$$

- comprehension or specification ("For all X and any formula F with free variable x, there is a set that we may call $\{x \in X \mid F(x)\}$.")
$$\mathcal{C}_\forall\big(\,\forall X\ \exists C\ \forall z\ (z \in C \leftrightarrow (z \in X \wedge F(z)))\big)$$
for all formulas F with a least a free variable x; here $\mathcal{C}_\forall$ quantifies universally over all other free variables that occur in F
- replacement ("For all X and any term t with free variable x, there is a set that we may call $\{t(x) : x \in X\}$.")
$$\mathcal{C}_\forall\big(\forall X\ \exists R\ \forall z\ (z \in R \leftrightarrow (\exists x\ (x \in X \wedge z \doteq t(x))))\big)$$
for all terms t with at least a free variable x
- regularity (which we will skip here)
- infinity ("There is an infinite set.")
$$\exists I\ (\exists x\ x \in I \wedge \forall x\ (x \in I \rightarrow \exists y\ (y \in I \wedge x \in y)))$$

Note that ZF is not a finite theory. It contains infinitely many instances of comprehension and replacement because they exist for any formula F or term t. These are often called axiom *schemas* to emphasize that they are actually infinite families of axioms.

The spirit of the axioms is as follows. Extensionality defines equality. Empty gets us off the ground by making sure that the simplest set exists; then pairing, union, and power set let us form bigger sets from existing ones. Comprehension goes backwards: If we have constructed a big set from small ones, comprehension gives us all the smaller ones in between. Replacement goes sideways: If we have constructed a set, it gives us one of the same size (or smaller) but with different elements.

Regularity avoids certain degenerate cases. In particular, regularity can be used to prove that there are no $\in$-cycles, i.e.,
$$\neg\exists x_1\ \ldots \exists x_n\ (x_1 \doteq x_n \wedge x_1 \in x_2 \wedge \ldots \wedge x_{n-1} \in x_n)$$
is a theorem for any $n = 1, 2, \ldots$.

Inspecting the previous axioms shows that they all preserve finiteness: From finite sets, we can construct only finite ones. The axiom of infinity is needed to get an infinite set. Here infinity is achieved by postulating a set containing $x_1,\ x_2,\ \ldots$ such that $x_1 \in x_2 \in \ldots$. Because of regularity, this captures our intuition about countable infinity. Then sets of other cardinalities can be constructed using the power set axiom.

Remark 16.2. Note that ZF has no non-trivial terms: The only terms are variables. However, other terms are introduced later using definition principles.

Remark 16.3. Our version of the replacement axiom scheme is not standard. It is usually stated with a formula $F(x, y)$ instead of a term $t(x)$. Then the intuition is that it gives us the set $\{y : x \in X,\ F(x, y)\}$. Our version arises from the more intuitive special case where we have $\forall x\ F(x, t(x))$.

Note that our version is pointless if we do not have any non-trivial terms. Therefore, it is usually stated with a formula instead of a term. Our version makes sense in light of Rem. 16.2.

There are many variants of ZF-style set theories using slightly different but equivalent axioms. In fact, ours is a new variant itself. It is usually not relevant which variant is used because they are all isomorphic in the sense that we have $\mathcal{FOLEQ}$-theory isomorphisms between them. However, one variant is important: the axiom of choice:

Definition 16.4. ZFC arises from ZF by adding the axiom of choice, which we will only state informally: For any set X of pairwise disjoint non-empty sets, there is a set C such that C contains exactly one element from every element of X.

In particular, if we think of X as the set of classes of an equivalence relation, then we can think of C as a system of representatives. So the axiom of choice lets us choose a system of representatives for any equivalence relation.

The main objection to the axiom of choice is that it has many weird consequences such as the Banach-Tarski paradox [**?**].

16.2 Definition Principles

ZF is a very weird FOLEQ theory because it does not have any function symbols and thus no terms except variables. Therefore, technically, we do not have any mathematical objects — we only have theorem saying that certain objects exist, perhaps uniquely. Every mathematical statement has to be coded in terms of formulas and variables.

Any set must be coded as a property $A(x)$ together with a proof of $\exists x\ \forall z\ (z \in A \leftrightarrow A(z))$. Then, for example, a function from $A(x)$ to $B(x)$ is a formula $F(x,y)$ together with a proof of $\forall x\ (A(x) \rightarrow \exists^! y\ B(y) \wedge F(x,y))$. Here $\exists^! y\ P(y)$ is the quantifier of unique existence defined as $\exists y\ (P(y) \wedge \forall z\ (P(z) \rightarrow z \doteq y))$.

This is, of course, awkward. So what do mathematicians do instead? They use definition principles to introduce function symbols.

Definition 16.5 (Direct Definition)**.** Consider a term t with free variables $x_1, \ldots, x_n$. Then appealing to the direct definition principle means to add a new n-ary function symbol f together with an axiom

$$\forall x_1\ \ldots \forall x_n\ f(x_1, \ldots, x_n) \doteq t(x_1, \ldots, x_n)$$

Similarly, consider a formula F with free variables $x_1, \ldots, x_n$. Then appealing to the direct definition principle means to add a new n-ary predicate symbol p together with an axiom

$$\forall x_1\ \ldots \forall x_n\ p(x_1, \ldots, x_n) \leftrightarrow F(x_1, \ldots, x_n)$$

The direct definition principle is obviously justified: f is simply an abbreviation for t, and any occurrence of $f(t_1, \ldots, t_n)$ can be replaced with $t[x_1/t_1, \ldots, x_n/t_n]$. The same argument holds for predicate symbols, although here some reasoning about equivalence is necessary.

Definition 16.6 (Description Principle)**.** Consider a formula $F(x_1, \ldots, x_n, y)$ with $n+1$ free variables together with a proof of

$$\forall x_1\ \ldots \forall x_n\ \exists^! y\ F(x_1, \ldots, x_n, y)$$

Then appealing to the description principle means to add a new n-ary function symbol f together with an axiom

$$\forall x_1\ \ldots \forall x_n\ F(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$$

Note that the action of the new function symbol f is uniquely determined (described) by F: $f(x_1, \ldots, x_n)$ is the unique object y such $F(x_1, \ldots, x_n, y)$.

We can eliminate f from any formula G by first replacing $f(t_1, \ldots, t_n)$ with a fresh variable y resulting in the formula G', and then forming the formula $G'' = \forall y\ (F(t_1, \ldots, t_n, y) \rightarrow G')$. Then we prove that G and G'' are equivalent.

Example 16.7. Using the axioms of ZF, it is easy to prove the unique existence of the unordered pair:

$$\forall X\ \forall Y\ \exists^! P\ \forall z\ (z \in P \leftrightarrow (z \in X \vee z \in Y))$$

Using the description principle, we obtain a binary function symbol f, and we denote $f(t, t')$ as $\{t, t'\}$

Similarly, we proceed with the axioms of union, power set, comprehension, and replacement introducing new function symbols and notations as already indicated in Def. 16.1. Note that the two axiom schemas introduce infinite families of function symbols. For example, comprehension introduces a unary function symbol $f_{F(x)}$ for every formula $F(x)$, and we introduce the notation $\{x \in A \,|\, F(x)\}$ for $f_{F(x)}(A)$.

The description principle is accepted by virtually everybody. It already gets complicated when we generalize a bit:

Definition 16.8 (Choice Principle)**.** Consider a formula $F(x_1, \ldots, x_n, y)$ with $n+1$ free variables together with a proof of

$$\forall x_1\ \ldots \forall x_n\ \exists y\ F(x_1, \ldots, x_n, y)$$

Then appealing to the choice principle means to add a new n-ary function symbol f together with an axiom

$$\forall x_1 \ \dots \forall x_n \ F(x_1, \dots, x_n, f(x_1, \dots, x_n))$$

The justification of the choice principle is well-known in logic, where it is already used in the Skolemization operation. The main objection to the choice principle is that we do not know what the result of $f(t_1, \dots, t_n)$ is. It feels wrong to give something a name that is not precisely described.

Remark 16.9. The choice principle and the choice axiom are closely connected and often confused, but the distinction is important. Typically, the choice axiom is used crucially in the proof of the precondition of the choice principle.

For example, the choice axiom guarantees the existence of a system of representatives, which we can think of a unary function mapping equivalence classes to their representatives. Then the choice principle lets us add a unary function symbol explicitly giving us the representatives.

The choice axiom and the choice principle are independent of each other: A separate leap of faith is required for either one, and each one make sense by itself even if one rejects the other one. The axiom makes new objects exist, the principle gives us new names for existing but undescribable objects.

Chapter 17

Higher-Order Logic

The variety of type theories is even bigger than that of set theories: The differences between simple type theory, dependent type theory, and the calculus of constructions are quite big. All three can be used as pure type theories (i.e., without formulas), as logics, and as foundations. We will only consider HOL, which is based on simple type theory.

17.1 Foundational Logic and Theory

In general, every symbol or axiom introduced in a foundational theory could alternatively be built into the foundational logic a priori. Thus, the formal border between foundational logic and foundational theory can be a bit blurry. This is specifically the case for HOL where different accounts vary regarding which features are present in HOL the logic and which are relegated to individual theories. The name "HOL" is commonly used for all of them.

To avoid confusion, we will reserve the name HOL for "HOL the logic" as defined in Part II

The foundational theory HOLF arises by adding choice and infinity to HOL. As mentioned above, there is some flexibility as to whether these features are hard-coded into HOL already or not. Our choice here tries to put all features that have a more mathematical flavor rather than a logical one into the foundational theory.

Definition 17.1. HOLF is the HOL-theory that consists of

- individuals: a single base type $\iota : \texttt{type}$ (in addition to the base type o which is built into HOL),
- description operator: a family of constants $\delta_A : (A \to o) \to A$ for any HOLF-type A,
- description axiom: the axiom schema

  $$\forall p : A \to o.(\exists^! x : A.p\ x \Rightarrow p\ (\delta_A\ p))$$

  for any HOLF type A,
- choice operator: a family of constants $\varepsilon_A : (A \to o) \to A$ for any HOLF-type A,
- choice axiom: the axiom schema

  $$\forall p : A \to o.(\exists x : A.p\ x \Rightarrow p\ (\varepsilon_A\ p))$$

  for any HOLF type A,
- infinity: an axiom forcing the infinity of ι, e.g., by stating the Peano axioms for ι.

Notation 17.2. We write $\delta x : A.p(x)$ for $\delta_A\ (\lambda x : A.p(x)))$, and $\varepsilon x : A.p(x)$ for $\varepsilon_A\ (\lambda x : A.p(x)))$.

The choice operator is the characteristic feature of the HOL foundation. If $p : A \to o$ is a predicate on A, then $\varepsilon x : A.p(x)$ returns some element of A satisfying p. There are two objections to this feature. Firstly, $\varepsilon_A\ p$ is well-formed even if there is no such element. Indeed, $\varepsilon x : \iota.\mathit{false}$ is a well-formed term. Secondly, it is left unspecified what the result of $\varepsilon_A\ p$ is if there is more than x with the require property. For the description operator, only the first objection applies.

HOLF does not run into inconsistencies due to these problems because the corresponding axioms are chosen carefully: It only states the property of $\varepsilon x : A.p(x)$ (namely that it satisfies p) if p is satisfiable at all, and then all we know about it is that it satisfies p. Otherwise, the result of $\varepsilon x : A.p(x)$ is unspecified. Thus, we can prove that $\varepsilon x : \iota.\mathit{false}$ exists but nothing else about it. Therefore, as far as we know, the only tangible side effect of the choice operator is that all types are non-empty.

The description operator is a special case of the choice operator and therefore redundant. But because it is less objectionable, it is preferable to use it whenever that is sufficient. In fact, HOLF is often formulated without the choice operator because the description operator is enough for most applications.

17.2 Definition Principles

Like virtually every foundation ever defined HOLF admits the analogue of the direct definition principle from Def. 16.5. However, due to the presence of λ-abstraction and because o is just a normal type, the definition principle takes a significantly simpler form:

Definition 17.3 (Direct Definition)**.** For any well-formed closed term $t : A$, we may add a new constant $c : A$ and an axiom $c \doteq t$.

While direct definitions exists in some form in any foundations, the second definition principle is a bit peculiar and specific to HOL:

Definition 17.4 (Type Definition)**.** For any well-formed closed term $p : A \to o$ together with a proof of $\exists x : A.p\ x$, we may add

- a new type $t : \mathtt{type}$,
- constants $r : t \to A$ and $a : A \to t$,
- the axiom $\forall x : t.p\ (r\ x)$,
- the axiom $\forall x : t.a\ (r\ x) \doteq x$,
- the axiom $\forall x : A.(p\ x \Rightarrow r\ (a\ x) \doteq x)$.

The intuition behind this extension is that the new type t represents the subtype of A containing those objects that satisfy p. Since HOL cannot directly describe this type, a new base type t is introduced and r and a map back and forth between A and t. These map is axiomatized such that t becomes indeed isomorphic to the subtype.

This type definition is strong enough to define most useful types as derived notions, such as record types and inductive types.

Chapter 18

Implemented Foundations

The languages for proofs in the foundation can be informal or formal. Mathematicians tend to use set theory and then favor informal proofs that use natural – albeit highly standardized and precise – language. Type theory tends to favor a more restricted language of propositions and a completely formal language of proofs.

Formal proofs have the advantage that they can be written, verified, and to some extent even found by computers. There have been several major research projects aiming at developing mathematics fully formally using an implementation of a foundation. Almost all of them use a type theoretic foundation because they have better computational properties than set theories.

All formal foundations are far away from covering all of mathematics. Most of the time the formalized proofs reach about undergraduate or graduate level mathematics. However, the amount is growing, and several large-scale projects are in progress. Most notably the Flyspeck project [?] (using HOL Light) and the formalization of the classification theorem for simple groups [?] (using Coq).

The biggest bottleneck in using formal foundations is the de-Bruijn factor: the factor by which a formal proof takes more space and more time to write than the corresponding informal proof. Both can be up to 10 or 20 and are at the moment nowhere close to 1, let alone smaller than 1. Therefore, they are rarely used by mathematicians. But they are used increasingly in computer science, specifically in the verification of software and hardware.

In order to cut down the work in writing proofs, these tools usually come with sophisticated proof development languages. These often provide key words for structured natural deduction style proofs, permit the use of small programs that compute proofs (tactics, often coming with their own tactic programming language), or employ external tools to simplify expressions and proof goals.

Automath Automath [?] was the first major project in this direction. It employed a logical framework so that different formal foundations could be declared. Thus, the system itself could be foundationally uncommitted. The biggest experiment in this system was the formalization of Landau's "Foundations of Analysis" text book [?]. Automath is hardly in use anymore today.

Mizar Mizar is the only large-scale formal foundation based on set theory. It uses the Tarski-Grothendieck variant based on FOLEQ, which is very close to ZF. Mizar adds three characteristic features: a large variety of additional definition principles, e.g., for case-based function symbols; a flexible type system (with dependent types but without function types) that permits users to employ typed reasoning within the untyped language; and a high-level language that is tuned to be as close to informal mathematical language as possible. The Mizar library comprises about 50000 theorems and is the largest at the moment.

The HOL Family There are several implementations of slightly varying HOL-based foundations, most importantly HOL [?], Isabelle/HOL [?], and HOL Light [?]. Isabelle/HOL uses a logical framework (which is itself based on a variant of HOL) and generic theorem prover [?], and then defines HOL within it.

The Calculus of Constructions Family Coq [?] is based on the calculus of constructions, which is so expressive that it can be used as a foundation right away or as a logical framework in which other foundations are defined. Matita [?] is a Coq-inspired reimplementation.

Undecidable Type Theories PVS [?] and Nuprl [?] are based on very rich type theoretical foundations, where typing is undecidable.

Part IV

Logic Translations

Chapter 19

Translating First-Order to Higher-Order Logic

In this chapter, we will define a tuple $(\Phi, \alpha, \beta, \gamma)$ constituting a logic translation from FOLEQ to HOL. The four components will be defined individually. Moreover, each of them has two components, one dealing with signatures, one dealing with signature morphisms:

19.1 The Object Dimension

Definition 19.1 (Signature Translation)**.** Given a FOLEQ signature $\Sigma = (\Sigma_f, \Sigma_p, ar)$, the HOL signature $\Phi(\Sigma)$ contains the following declarations:

- $\iota : \mathtt{type}$
- $f : \underbrace{\iota \to \ldots \to \iota}_{n} \to \iota$ for every $f \in \Sigma_f$ with $ar(f) = n$
- $p : \underbrace{\iota \to \ldots \to \iota}_{n} \to o$ for every $p \in \Sigma_p$ with $ar(p) = n$

Definition 19.2 (Expression Translation)**.** Given $\Gamma \vdash_\Sigma t : \mathtt{term}$ or $\Gamma \vdash_\Sigma F : \mathtt{form}$, we define $\alpha_\Sigma(t)$ and $\alpha_\Sigma(F)$ by induction of Γ, t, and F. We omit the details.

Definition 19.3 (Judgment and Proof Translation)**.** Given a Σ-judgment $J \in \mathbf{Pf}^{\mathcal{FOLEQ}}(\Sigma)$ (see Def. 4.5), the $\Phi(\Sigma)$-judgment $\gamma_\Sigma(J)$ is defined as follows:

$$\begin{cases} x_1 : \iota, \ldots, x_n : \iota \vdash_{\Phi(\Sigma)} \alpha_\Sigma(t) \ : \ \iota & \text{if } J \ = \ x_1, \ldots, x_n \vdash_\Sigma F : \mathtt{term} \\ x_1 : \iota, \ldots, x_n : \iota \vdash_{\Phi(\Sigma)} \alpha_\Sigma(F) \ : \ o & \text{if } J \ = \ x_1, \ldots, x_n \vdash_\Sigma F : \mathtt{form} \\ \alpha_\Sigma(F_1), \ldots, \alpha_\Sigma(F_n); x_1 : \iota, \ldots, x_n : \iota \vdash_{\Phi(\Sigma)} \alpha_\Sigma(F) & \text{if } J \ = \ F_1, \ldots, F_m; x_1, \ldots, x_n \vdash_\Sigma F \end{cases}$$

Given a Σ-proof p of a judgment J using assumptions $J_1, \ldots, J_m$, the $\Phi(\Sigma)$-proof $\gamma_\Sigma(p)$ of $\gamma_\Sigma(J)$ using assumptions $\gamma_\Sigma(J_1), \ldots, \gamma_\Sigma(J_m)$ is obtained by induction of p. We omit the details.

Remark 19.4. Note that the second part of this definition includes a substantial theorem: By mapping every Σ-proof of J to a $\Phi(\Sigma)$-proof of $\gamma_\Sigma(J)$, we show that all proofs are translated. In particular, if J is provable over Σ, then so is $\gamma_\Sigma(J)$ over $\mathbf{Pf}^{\mathcal{HOL}}(\Phi(\Sigma))$.

Definition 19.5 (Model and Model Morphism Translation)**.** Given a Σ-model $I \in \mathbf{Mod}^{\mathcal{FOLEQ}}(\Phi(\Sigma))$, the Σ-model $\beta_\Sigma(I)$ is defined as follows: [1]
Given a Σ-model morphism $m : I \to I'$ in $\mathbf{Mod}^{\mathcal{FOLEQ}}(\Phi(\Sigma))$, the Σ-model morphism $\beta_\Sigma(m) : \beta_\Sigma(I) \to \beta_\Sigma(I')$ is defined as follows:

19.2 The Morphism Dimension

[2]

Definition 19.6 (Signature Morphism Translation)**.** Given a FOLEQ signature morphism $\sigma : \Sigma \to \Sigma'$, the HOL signature morphism $\Phi(\sigma) : \Phi(\Sigma) \to \Phi(\Sigma')$ maps as follows:

- $\iota \mapsto \iota$
- $f \mapsto \lambda x_1 : \iota. \dots \lambda x_n : \iota.\sigma(f)$ for every $f \in \Sigma_f$ with $ar(f) = n$
- $p \mapsto \lambda x_1 : \iota. \dots \lambda x_n : \iota.\sigma(p)$ for every $p \in \Sigma_p$ with $ar(p) = n$

Recall that Def. 9.1 makes $\sigma(f)$ and $\sigma(p)$ expressions with free variables $x_1, \dots, x_n$. Those can now be properly bound because we have λ-abstraction in HOL.

Now consider a FOLEQ-signature morphism $\sigma : \Sigma \to \Sigma'$ and an expression E over Σ. There are now two ways to translate E to an expression over $\Phi(\Sigma')$, depending on whether we first translate along σ or first along α:

$$\begin{tikzcd}
\Sigma\text{-expressions} \arrow[r, "\alpha_\Sigma"] \arrow[d, "\overline{\sigma}"'] & \Phi(\Sigma)\text{-expressions} \arrow[d, "\overline{\Phi(\sigma)}"'] \\
\Sigma'\text{-expressions} \arrow[r, "\alpha_{\Sigma'}"] & \Phi(\Sigma')\text{-expressions}
\end{tikzcd}$$

In particular, recalling that $\mathbf{Sen}(\Sigma)$ contains those expressions that are sentences and that $\mathbf{Sen}(\sigma)$ is the restriction of $\overline{\sigma}$ to $\mathbf{Sen}(\Sigma)$:

$$\begin{tikzcd}
\mathbf{Sen}(\Sigma) \arrow[r, "\alpha_\Sigma"] \arrow[d, "\mathbf{Sen}(\sigma)"'] & \mathbf{Sen}'(\Phi(\Sigma)) \arrow[d, "\mathbf{Sen}'(\Phi(\sigma))"'] \\
\mathbf{Sen}(\Sigma') \arrow[r, "\alpha_{\Sigma'}"] & \mathbf{Sen}'(\Phi(\Sigma'))
\end{tikzcd}$$

It turns out, it these two are equal:

Lemma 19.7 (Commutativity of Φ and α)**.** *For all $\sigma : \Sigma \to \Sigma$ in $\mathbf{Sig}^{\mathcal{FOLEQ}}$, we have $\overline{\Phi(\sigma)} \circ \alpha_\Sigma = \alpha_{\Sigma'} \circ \overline{\sigma}$.*

A similar situation arises for the judgment and proof translation:

$$\begin{tikzcd}
\mathbf{Pf}^{\mathcal{FOLEQ}}(\Sigma) \arrow[r, "\gamma_\Sigma"] \arrow[d, "\overline{\sigma}"'] & \mathbf{Pf}^{\mathcal{HOL}}(\Phi(\Sigma)) \arrow[d, "\overline{\Phi(\sigma)}"'] \\
\mathbf{Pf}^{\mathcal{FOLEQ}}(\Sigma') \arrow[r, "\gamma_{\Sigma'}"] & \mathbf{Pf}^{\mathcal{HOL}}(\Phi(\Sigma'))
\end{tikzcd}$$

[2]This whole section can be skipped.

Again it turns out, both are equal:

Lemma 19.8 (Commutativity of Φ and γ)**.** *For all $\sigma : \Sigma \to \Sigma$ in $\mathbf{Sig}^{\mathcal{FOLEQ}}$, we have $\overline{\Phi(\sigma)} \circ \gamma_\Sigma = \gamma_{\Sigma'} \circ \overline{\sigma}$ both for judgments and for proofs.*

Finally, the corresponding situation – but with reversed directions – arises for the model translation:

$$\begin{tikzcd}
\mathbf{Mod}^{\mathcal{FOLEQ}}(\Sigma) & \mathbf{Mod}^{\mathcal{HOL}}(\Phi(\Sigma)) \arrow[l, "\beta_\Sigma"'] \\
\mathbf{Mod}^{\mathcal{FOLEQ}}(\Sigma') \arrow[u, "-|_\sigma"] & \mathbf{Mod}^{\mathcal{HOL}}(\Phi(\Sigma')) \arrow[l, "\beta_{\Sigma'}"'] \arrow[u, "-|_{\Phi(\sigma)}"]
\end{tikzcd}$$

Again it turns out, both are equal:

Lemma 19.9 (Commutativity of Φ and β)**.** *For all $\sigma : \Sigma \to \Sigma$ in $\mathbf{Sig}^{\mathcal{FOLEQ}}$, we have $\beta_\Sigma \circ -|_{\Phi(\sigma)} = -|_\sigma \circ \beta_{\Sigma'}$ both for models and model morphisms.*

19.3 Invariants

As pointed out in Rem. 19.4, the existence of γ already guarantees that provability is preserved. Moreover, we observe:

Lemma 19.10 (Commutativity of α and γ)**.** *For all $\Sigma \in \mathbf{Sig}^{\mathcal{FOLEQ}}$ and all $F \in \mathbf{Sen}^{\mathcal{FOLEQ}}(\Sigma)$, we have*

$$\gamma_\Sigma(\vdash^{\mathcal{FOLEQ}}_\Sigma F) = \vdash^{\mathcal{HOL}}_{\Phi(\Sigma)} (\alpha_\Sigma(F)).$$

Proof. Directly from the definitions. □

Now we know that the validity of FOLEQ is translated to the validity judgment of HOL. Together with the proof-preservation of γ, we obtain the central theorem:

Notation 19.11. For $\Theta \subseteq \mathbf{Sen}(\mathbf{Sen}(\Sigma)$, we write $\alpha_\Sigma(\Theta)$ for $\{\alpha_\Sigma(F) : F \in \Theta\}$.

Theorem 19.12 (Preservation of Proof-Theoretical Consequence)**.** *For all $\mathcal{FOLEQ}$-theories (Σ, Θ) and all $F \in \mathbf{Sen}(\Sigma)$, if F is a proof theoretical theorem of (Σ, Θ), then $\alpha_\Sigma(F)$ is a proof theoretical theorem of $(\Phi(\Sigma), \alpha_\Sigma(\Theta))$.*

Proof. Immediate. □

Lemma 19.13 (Commutativity of α and β)**.** *For all $\Sigma \in \mathbf{Sig}^{\mathcal{FOLEQ}}$ and all $F \in \mathbf{Sen}^{\mathcal{FOLEQ}}(\Sigma)$, we have*

$$\beta_\Sigma(I) \models^{\mathcal{FOLEQ}}_\Sigma F \quad \text{iff} \quad I \models^{\mathcal{HOL}}_{\Phi(\Sigma)} \alpha_\Sigma(F) \quad \text{for all } M \in \mathbf{Mod}^{\mathcal{HOL}}(\Phi(\Sigma)).$$

Now we know that the satisfaction relation of FOLEQ is translated to the satisfaction relation of HOL. This yields the central theorem:

Theorem 19.14 (Preservation of Model-Theoretical Consequence)**.** *For all $\mathcal{FOLEQ}$-theories (Σ, Θ) and all $F \in \mathbf{Sen}(\Sigma)$, if F is a model theoretical theorem of (Σ, Θ), then $\alpha_\Sigma(F)$ is a model theoretical theorem of $(\Phi(\Sigma), \alpha_\Sigma(\Theta))$.*

Proof. Assume $I \in \mathbf{Mod}^{\mathcal{HOL}}(\Phi(\Sigma))$ and $I \models^{\mathcal{HOL}}_{\Phi(\Sigma)} \alpha_\Sigma(A)$ for all $A \in \Theta$ (*). We need to show $I \models^{\mathcal{HOL}}_{\Phi(\Sigma)} \alpha_\Sigma(F)$. (*) and Lem. 19.13 yield $\beta_\Sigma(I) \models^{\mathcal{FOLEQ}}_\Sigma A$ for all $A \in \Theta$. Therefore, using that F is a model theoretical theorem of (Σ, Θ), we infer $\beta_\Sigma(I) \models^{\mathcal{FOLEQ}}_\Sigma F$. Using Lem. 19.13 again, we obtain the needed property. □

Chapter 20

An Abstract Definition of a Logic Translation

Now we collect the definitions from Sect. 19 and abstract from them. As before, we distinguish advanced material that can be ignored.

20.1 Translations

Definition 20.1 (Syntax)**.** Given two logic syntaxes $(\mathbf{Sig}, \mathbf{Sen})$ and $(\mathbf{Sig}', \mathbf{Sen}')$, we define a logic syntax translation as a pair (Φ, α) such that

- Φ is a mapping
  - $\mathbf{Sig} \to \mathbf{Sig}'$
  - $\mathbf{Sig}(\Sigma_1, \Sigma_2) \to \mathbf{Sig}'(\Phi(\Sigma_1), \Phi(\Sigma_2))$ for all $\Sigma, \Sigma' \in \mathbf{Sig}$
- α is a family of mappings α_Σ for $\Sigma \in \mathbf{Sig}$ such that
  - $\alpha_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\Phi(\Sigma))$
  - for all $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sig}$, we have $\mathbf{Sen}'(\Phi(\sigma)) \circ \alpha_\Sigma = \alpha_{\Sigma'} \circ \mathbf{Sen}(\sigma)$

Definition 20.2 (Proof Theory)**.** Given two proof theoretical logics $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash)$ and $(\mathbf{Sig}', \mathbf{Sen}', \mathbf{Pf}', \vdash')$, and given a syntax translation (Φ, α) between them, a proof theory translation γ

- is a family of mappings γ_Σ for $\Sigma \in \mathbf{Sig}$ such that
  - γ_Σ is a mapping such that
    * $\gamma_\Sigma : \mathbf{Pf}(\Sigma) \to \mathbf{Pf}'(\Phi(\Sigma))$
    * $\gamma_\Sigma : \mathbf{Pf}(\Sigma)((J_1, \ldots, J_m), J) \to \mathbf{Pf}'(\Phi(\Sigma))((\gamma_\Sigma(J_1), \ldots, \gamma_\Sigma(J_m)), \gamma_\Sigma(J))$
  - for all $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sig}$, we have $\mathbf{Pf}'(\Phi(\sigma)) \circ \gamma_\Sigma = \gamma_{\Sigma'} \circ \mathbf{Pf}(\sigma)$
- such that for all $\Sigma \in \mathbf{Sig}$ and all $F \in \mathbf{Sen}(\Sigma)$

$$\gamma_\Sigma(\vdash_\Sigma (F)) = \vdash'_{\Phi(\Sigma)} (\alpha_\Sigma(F))$$

Definition 20.3 (Model Theory)**.** Given two model theoretical logics $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ and $(\mathbf{Sig}', \mathbf{Sen}', \mathbf{Mod}', \models')$, and given a syntax translation (Φ, α) between them, a model theory translation

γ

- is a family of mappings β_Σ for $\Sigma \in \mathbf{Sig}$ such that
  - β_Σ is a mapping such that
    - $\beta_\Sigma : \mathbf{Mod}'(\Phi(\Sigma)) \to \mathbf{Mod}(\Sigma)$
    - $\beta_\Sigma : \mathbf{Mod}'(\Phi(\Sigma))(M, N) \to \mathbf{Mod}(\Sigma)(\beta_\Sigma(M), \beta_\Sigma(N))$
  - for all $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sig}$, we have $\beta_\Sigma \circ \mathbf{Mod}'(\Phi(\Sigma)) = \mathbf{Mod}(\sigma) \circ \beta'_\Sigma$
- such that for all $\Sigma \in \mathbf{Sig}$ and all $F \in \mathbf{Sen}(\Sigma)$

$$\beta_\Sigma(M) \models_\Sigma F \qquad \text{iff} \qquad M \models'_{\Phi(\Sigma)} \alpha_\Sigma(F) \qquad \text{for } M \in \mathbf{Mod}'(\Phi(\Sigma))$$

Remark 20.4. Model theoretical logics are usually called *institutions*. Model theory translations between institutions are usually called *institution comorphisms*. Institution morphisms also exists. Arguably, the names of morphisms and comorphisms should have better been swapped in this case.

Definition 20.5 (Logics)**.** Given two logics $L = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$ and $L' = (\mathbf{Sig}', \mathbf{Sen}', \mathbf{Mod}', \models', \mathbf{Pf}', \vdash')$, a *logic translation* from L to L' consists of

- a syntax translation (Φ, α) from $(\mathbf{Sig}, \mathbf{Sen})$ to $(\mathbf{Sig}', \mathbf{Sen}')$,
- a model theory translation β extending (Φ, α),
- a proof theory translation γ extending (Φ, α).

20.2 Preservation of Consequence

In Sect. 19.3, we proved two invariants about the logic translation from FOL to HOL. Both hold in general:

Theorem 20.6 (Preservation of Proof-Theoretical Consequence)**.** *Consider a proof theory translation* $(\Phi, \alpha, \gamma) : (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash) \to (\mathbf{Sig}', \mathbf{Sen}', \mathbf{Pf}', \vdash')$*. Moreover, consider a theory* (Σ, Θ) *and a sentence* $F \in \mathbf{Sen}(\Sigma)$*. If* F *is a proof theoretical theorem of* (Σ, Θ)*, then* $\alpha_\Sigma(F)$ *is a proof theoretical theorem of* $(\Phi(\Sigma), \alpha_\Sigma(\Theta))$*.*

Proof. □

Theorem 20.7 (Preservation of Model-Theoretical Consequence)**.** *Consider a model theory translation* $(\Phi, \alpha, \beta) : (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models) \to (\mathbf{Sig}', \mathbf{Sen}', \mathbf{Mod}', \models')$*. Moreover, consider a theory* (Σ, Θ) *and a sentence* $F \in \mathbf{Sen}(\Sigma)$*. If* F *is a model theoretical theorem of* (Σ, Θ)*, then* $\alpha_\Sigma(F)$ *is a model theoretical theorem of* $(\Phi(\Sigma), \alpha_\Sigma(\Theta))$*.*

Proof. Assume $I \in \mathbf{Mod}'(\Phi(\Sigma))$ and $I \models'_{\Phi(\Sigma)} \alpha_\Sigma(A)$ for all $A \in \Theta$. We need to show $I \models'_{\Phi(\Sigma)} \alpha_\Sigma(F)$. By the properties of model theory translations, we obtain $\beta_\Sigma(I) \models_\Sigma A$ for all $A \in \Theta$. Therefore, using that F is a model theoretical theorem of (Σ, Θ), we infer $\beta_\Sigma(I) \models_\Sigma F$. Using the properties of model theory translations again, we obtain the needed property. □

Chapter 21

Model Theories as Logic Translations

21.1 Models as Theory Morphisms

In Sect. 15.3, we have seen that foundations consist of a fixed theory in a fixed proof theoretical logic (along with some definition principles). Now consider a logic syntax $L = (\mathbf{Sig}, \mathbf{Sen})$. We would like to define a model theory for it using a foundation consisting of P and $\mathcal{F}$. The key observation is that the inductive definition of the interpretation function $[\![-]\!]^I$ for a model I behaves very much like the inductive expression translation function $\overline{\sigma}$ obtained from a theory morphism σ: Both are an inductive, compositional mapping in which every expression constructor of L is mapped to a value constructor in mathematics. We want to make that precise and define models as theory morphisms. We can do that by using a special logic translation from L to P:

Definition 21.1. Assume a logic syntax $L = (\mathbf{Sig}, \mathbf{Sen})$ and a foundation $\mathcal{F} \in \mathbf{Th}^P$. A syntax translation $(\Phi, \alpha) : L \to P$ is called $\mathcal{F}$-founded if

- $\Phi(\Sigma)$ is an extension of $\mathcal{F}$ for every $\Sigma \in \mathbf{Sig}$
- $\Phi(\sigma)$ is the identity on $\mathcal{F}$ for every signature morphism σ

Definition 21.2. Assume a logic syntax $L = (\mathbf{Sig}, \mathbf{Sen})$, a foundation $\mathcal{F} \in \mathbf{Th}^P$, and an $\mathcal{F}$-founded syntax translation $(\Phi, \alpha) : L \to P$.
We define $\mathbf{Mod}(\Sigma)$ as the collection of P-theory morphisms from $\Phi(\Sigma)$ to $\mathcal{F}$ that are the identity on $\mathcal{F}$.
Moreover, if $\sigma : \Phi(\Sigma) \to \mathcal{F}$ is such a model and $F \in \mathbf{Sen}(\Sigma)$, we define

$$\sigma \models_\Sigma F \qquad \text{iff} \qquad \vdash^P_{\mathcal{F}} \overline{\sigma}(\alpha_\Sigma(F))$$

Remark 21.3. More generally, for the interpretation function of L-Σ-expressions in the model σ, we obtain

$$[\![-]\!]^\sigma = \overline{\sigma} \circ \alpha_\Sigma.$$

Remark 21.4. More generally, we could use morphisms from $\Phi(\Sigma)$ to any conservative extension of $\mathcal{F}$ in Def. 21.2.

Theorem 21.5. *Given a logic syntax* $(\mathbf{Sig}, \mathbf{Sen})$*, a foundation* $\mathcal{F} \in \mathbf{Th}^P$*, and an* $\mathcal{F}$*-founded syntax translation* $(\Phi, \alpha) : L \to P$*. Then* $(\mathbf{Mod}, \models)$ *as defined in Def. 21.2 is a model theory for* $(\mathbf{Sig}, \mathbf{Sen})$.

Example 21.6. Consider $(\mathbf{Sig}^{\mathcal{FOLEQ}}, \mathbf{Sen}^{\mathcal{FOLEQ}})$ and the foundation $HOLF \in \mathbf{Th}^{\mathcal{HOL}}$. We define a translation (Φ, α) that is very similar to the one from Def. 19.1 and 19.1. There are only two difference: (i) $\Phi(\Sigma)$ additionally contains all the declarations of $HOLF$; (ii) $\Phi(\Sigma)$ contains an additional base type u, which serves as the translation of the $\mathcal{FOLEQ}$-universe (instead of ι).

For example, the $\mathcal{FOLEQ}$-signature of monoids is mapped to the $\mathcal{HOL}$ theory M that extends $HOLF$ with

- $u : \mathtt{type}$,
- $\circ : u \to u \to u$
- $e : u$

Now a model of that signature is a $\mathcal{HOL}$-theory morphism from M to $HOLF$ that is the identity on $HOLF$. Thus, every $HOLF$-symbol is mapped to itself, and we only have to map u, $\circ$, and e is some way. That is exactly what a model is supposed to do.

21.2 A More Abstract Formulation

In order to formulate the notion of "$\mathcal{F}$-founded translations" more elegantly, we can use two constructions. Both work for arbitrary logics; therefore, we state them in full generality (which actually makes them more elegant).

The Logic of Theories For every logic L, we can construct another logic $\mathbf{Th}(L)$ such that the signatures of $\mathbf{Th}(L)$ are the theories of L.

Definition 21.7. Given a logic $L = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$, we define the logic $\mathbf{Th}(L) = (\mathbf{Th}^L, \mathbf{Sen}', \mathbf{Mod}', \models', \mathbf{Pf}, \vdash')$ as follows:

- $\mathbf{Th}^L$ consists of the theories and theory morphisms of L,
- $\mathbf{Sen}'$ is like $\mathbf{Sen}$ but ignores the axioms: $\mathbf{Sen}'((\Sigma;\Theta)) = \mathbf{Sen}(\Sigma)$ and $\mathbf{Sen}'(\sigma) = \mathbf{Sen}(\sigma)$,
- $\mathbf{Mod}'(\Sigma;\Theta)$ is $\mathbf{Mod}(\Sigma;\Theta)$ from Def. 3.23,
- $\models'_{(\Sigma,\Theta)}$ is the same as $\models_\Sigma$,
- $\mathbf{Pf}'(\Sigma;\Theta)$ has the same judgments as $\mathbf{Pf}(\Sigma)$, and all proofs may additionally use $\vdash_\Sigma F$ as assumptions for all $F \in \Theta$,
- $\vdash'_{(\Sigma,\Theta)}$ is the same $\vdash_\Sigma$

The corresponding definition applies to proof or model theoretical logics.

The Logic of Extensions For every logic L and a fixed signature Σ, we can construct another logic $L^{\Sigma\hookrightarrow}$ that arises as as a sublogic of L: We only consider those signature that extend Σ.

Definition 21.8. Given a logic $L = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$ and a signature $\Sigma \in \mathbf{Sig}$, we define the logic $L^{\Sigma\hookrightarrow} = (\mathbf{Sig}', \mathbf{Sen}', \mathbf{Mod}', \models', \mathbf{Pf}, \vdash')$ as follows:

- $\mathbf{Sig}'$ consists of the signatures that extend Σ and those signature morphisms between them that are the identity on Σ,
- $\mathbf{Sen}'$, $\mathbf{Mod}'$, $\models'$, $\mathbf{Pf}'$ and $\vdash'$ are the appropriate restrictions of their counterparts in L

The corresponding definition applies to proof or model theoretical logics.

Remark 21.9. Technically, Def. 21.8 only applies to logics in which *signature inclusions* are a defined notion. This is not always the case, but almost always and almost always obviously so. For example, it is obvious what a signature inclusion in FOLEQ is.

Founded Translations Now we can combine the two constructions:

Lemma 21.10. *Assume a logic syntax* $(\mathbf{Sig}, \mathbf{Sen})$ *and a foundation* $\mathcal{F} \in \mathbf{Th}^P$*. An* $\mathcal{F}$*-founded syntax translation* $L \to P$ *is the same as a syntax translation* $L \to \mathbf{Th}(P)^{\mathcal{F}\hookrightarrow}$*.*

Example 21.11. Consider the logic $\mathbf{Th}(\mathcal{FOLEQ})^{ZFC\hookrightarrow}$. Its "signatures" are the FOLEQ-theories that extend ZFC.

Part V

A Proof-Theoretical Logical Framework

Chapter 22

LF for Everybody

LF [?, ?] is a logical framework, i.e., a formal language that we use to define other formal languages. It is a very advanced language, so advanced that we were only able to understand it in the 1980s. Therefore, it is very hard to learn and should only be attempted after gaining a solid basis of knowledge about logic. On the other hand, the intuitions underlying LF are strikingly simple and the precise definition of LF is actually quite short.

The major problem with learning LF is that LF goes together with a certain methodology of approaching logics. This methodology is so powerful and elegant that users of LF become very familiar with it, so much that they forget there is any other way to do it. Consequently, they often apply this methodology even when describing LF itself, thus scaring everybody off who has not embraced it yet. That makes it virtually impossible to learn LF on your own.

Therefore, we will go a different route here: We will ignore the foundations of LF altogether and focus on how we can use LF in practice. *The goal is to show that LF is simple, natural, and practical rather than – as many are inadvertently led to think – difficult, weird, and esoteric.*

22.1 The Context-Free Fragment

> Lesson 1: LF is just a tool that we use to write down grammars. For convenience, we give the productions names.

Grammars and Named Productions Consider the grammar of propositional logic over a fixed set Σ of propositional variables:

`form`	$::=$	*true*	*truth*
	$\mid$	*false*	*falsity*
	$\mid$	`form` $\wedge$ `form`	*conjunction*
	$\mid$	`form` $\vee$ `form`	*disjunction*
	$\mid$	`form` $\Rightarrow$ `form`	*implication*
	$\mid$	$\neg$`form`	*negation*
	$\mid$	p where $p \in \Sigma$	*boolean variables*

Moreover, assume $p, q \in \Sigma$ and consider this example of a syntax tree for the formula $(\mathit{true} \wedge p) \vee q$:

$$\begin{tikzcd}
& & \texttt{form} \arrow[dl] \arrow[d, dash, dashed] \arrow[dr] & \\
& \texttt{form} \arrow[dl] \arrow[d, dash, dashed] \arrow[dr] & \vee & \texttt{form} \arrow[d] \\
\texttt{form} \arrow[d] & \wedge & \texttt{form} \arrow[d] & q \\
\mathit{true} & & p &
\end{tikzcd}$$

As usual, every node corresponds to a subformula of $(true \wedge p) \vee q$ and is labelled with the non-terminal symbol with which the derivation of that subformula begins. In particular, the root corresponds to the whole formula and every leaf node to an atomic formula. Moreover, some nodes produce additional terminal symbols (given in gray).

The key property of a syntax tree is that every node is connected to its children via a production. With a tiny change, we are able to state that much more elegantly: We give the productions names.

Giving names to production is very natural, for example, we can use

$$
\begin{array}{lcl}
\texttt{form} & ::= & \mathsf{truth}\text{: } \#\ true \\
 & | & \mathsf{falsity}\text{: } \#\ false \\
 & | & \mathsf{conj}\text{: } \#\ \texttt{form} \wedge \texttt{form} \\
 & | & \mathsf{disj}\text{: } \#\ \texttt{form} \vee \texttt{form} \\
 & | & \mathsf{impl}\text{: } \#\ \texttt{form} \Rightarrow \texttt{form} \\
 & | & \mathsf{neg}\text{: } \#\ \neg\texttt{form} \\
 & | & V_p\ \#\ p \text{ where } p \in \Sigma
\end{array}
$$

Note that naming the productions actually helps documenting and reading the grammar.

Now that the productions have names, we can use them to label the nodes in the syntax tree:

$$\begin{tikzcd}
 & & \texttt{disj} \arrow[dl] \arrow[d, dashed] \arrow[dr] & \\
 & \texttt{conj} \arrow[dl] \arrow[d, dashed] \arrow[dr] & \vee & V_q \arrow[d] \\
\texttt{truth} \arrow[d] & \wedge & V_p \arrow[d] & q \\
true & & p &
\end{tikzcd}$$

Now we observe that the nodes for the terminal symbols are actually redundant. Indeed, if productions have names, we do not need terminal symbols (given in gray above) *in a syntax tree*. Just like brackets, the sole purpose of $\wedge$ and $\vee$ is as a visual aid to display formulas nicely *in a line*. And the terminal symbols like *true*, p, and q are redundant because they are determined by the name of the preceding production.

There is no commonly used convention how to write named productions in BNF. For example, we use an ad-hoc notation and write our grammar as follows:

$$
\begin{array}{lcl}
\texttt{form} & ::= & \texttt{truth}\ \# \\
 & | & \texttt{falsity}\ \# \\
 & | & \texttt{conj}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{disj}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{impl}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{neg}\ \#\ \texttt{form} \\
 & | & V_p\ \#\ \text{ where } p \in \Sigma
\end{array}
$$

This may look a bit unusual, but observe that it still contains the same information as our original grammar.

Grammars in LF Grammars with named productions can be written down in LF directly. The basic idea is that non-terminal symbols and (named) productions are introduced by one declaration each:

1. For a non-terminal symbol A, we write $A : \texttt{type}$.

2. For a production $A ::= \ \# A_1 \ \ldots A_n$, we write $P : A_1 \to \ldots \to A_n \to A$.

Our grammar becomes

```
form     :  type.
truth    :  form.
falsity  :  form.
conj     :  form → form → form.
disj     :  form → form → form.
impl     :  form → form → form.
neg      :  form → form.
V_p      :  form.                    for p ∈ Σ
```

In LF text files, → is written as −>.

One might argue that the notation with `type` and → is somewhat arbitrary and even weird. Indeed, other notations can be more intuitive. Just for comparison, this is how the same grammar is written in SML:

```
datatype form =
   truth
 | falsity
 | conj of form * form
 | disj of form * form
 | impl of form * form
 | neg  of form * form
 | V_p                          for p in Σ
```

Note how there is one datatype for each non-terminal and one constructor for each production.

The specific notation of LF is in fact chosen very carefully and is very intuitive. But we will understand why only after learning a lot more about type theory.

Syntax Trees in LF In LF, we cannot only write down grammars but also syntax trees. Our example tree from above is written in LF as

$$(\texttt{conj}\ (\texttt{disj}\ \texttt{truth}\ V_p)\ V_q).$$

We only need brackets and whitespace to make sure that this linear representation can be uniquely translated back into a syntax tree.

This way of writing trees goes back to Lisp's S-expressions. It's also how SML does it. Some people might prefer the equivalent

$$\texttt{conj}(\texttt{disj}(\texttt{truth}, V_p), V_q).$$

If we want to use XML, we can write a syntax tree like in OpenMath:

```
<OMA>
  <OMS name="conj"/>
  <OMA><OMS name="disj"/><OMS name="truth"/><OMS name="V_p"/></OMA>
  <OMS name="V_q"/>
</OMA>
```

The bottom line is: We're now able to talk about syntax trees as objects! Instead of having to treat formulas as strings, we can treat them as syntax trees.

22.2 The Context-Sensitive Fragment

Lesson 2: It's useful to have productions that recurse into expressions with holes. LF supports that elegantly.

Let's first understand what we mean by "expressions with holes".

22.2.1 Expressions with Holes

Leaving Holes Consider our grammar for first-order logic:

`form`	$::=$	*true*	*truth*
	$\mid$	*false*	*falsity*
	$\mid$	`form` $\wedge$ `form`	*conjunction*
	$\mid$	`form` $\vee$ `form`	*disjunction*
	$\mid$	`form` $\rightarrow$ `form`	*implication*
	$\mid$	$\neg$`form`	*negation*
	$\mid$	$\forall$`var` `form`	*universal quantification*
	$\mid$	$\exists$`var` `form`	*existential quantification*
	$\mid$	`term` $\doteq$ `term`	*equality*
	$\mid$	$p(\overbrace{\texttt{term},\ldots,\texttt{term}}^{n})$ where $p \in \Sigma_p$ and $ar(p) = n$	*atomic formulas*
`term`	$::=$	$f(\overbrace{\texttt{term},\ldots,\texttt{term}}^{n})$ where $f \in \Sigma_f$ and $ar(f) = n$	*constants and function terms*
	$\mid$	`var`	*variables*
`var`	$::=$	some countable set of names	*variable names*

There is something fishy about this grammar, namely the non-terminal `var` with its vague production. This happens because – contrary to propositional logic – first-order logic is a language with binding, e.g., the x is bound in $\forall x\ F$.

Just like the named productions above, we can obtain a more elegant treatment of binding with a small conceptual leap: We introduce expressions with holes.

For example, let F be the formula $f(x, y) \doteq g(y)$. F has two variables – x and y – which we think of as placeholders for arbitrary terms. Let's make the placeholders explicit in the syntax tree:

$$\begin{tikzcd}
 & & x, y & \\
 & & \texttt{equal} \arrow[dl] \arrow[dr] & \\
 & V_f \arrow[dl] \arrow[dr] & & V_g \arrow[d] \\
\boxed{x} & & \boxed{y} & \boxed{y}
\end{tikzcd}$$

The placeholder nodes are boxed to indicate that they do not represent productions but unknown subtrees that will be determined later. To emphasize the holes in an expression, we will always list them above the root of the syntax tree.

Filling Holes Holes are placeholders for unknown expressions. Once known, we can plug in any expression for such a place holder. For example, in the expression F above, we can plug the syntax tree for $f(c)$ into the hole y:

$$\begin{tikzcd}
 & & x & \\
 & & \texttt{equal} \arrow[dl] \arrow[dr] & \\
 & V_f \arrow[dl] \arrow[dr] & & V_g \arrow[d] \\
\boxed{x} & & V_f \arrow[d] & V_f \arrow[d] \\
 & & V_c & V_c
\end{tikzcd}$$

Note that it is important that the holes are named: If a hole is used multiple times, we must plug in the same subtree everywhere. Also note that the resulting syntax tree only has one hole x.

Expressions with Holes in General Of course, plugging in expressions for holes is exactly what we know already from first-order logic as the substitution of expressions for free variables. But note that we can use expressions with named holes *for any grammar*. For example, let G be the expression $a \Rightarrow (b \wedge a)$ of propositional logic with two holes a and b; its syntax tree is

$$\begin{tikzcd}
 & a, b & & \\
 & \texttt{impl} \arrow[dl] \arrow[dr] & & \\
\boxed{a} & & \texttt{conj} \arrow[dl] \arrow[dr] & \\
 & \boxed{b} & & \boxed{a}
\end{tikzcd}$$

Note that we cannot generally plug in any subtree for any placeholder. For example, in F above, we can only plug in terms for x and y, i.e., a subtree whose root production is for the non-terminal `term`. We call `term` the *type* of the placeholders x and y.

Let H be the expression, $(f(x, y) \doteq g(y)) \wedge a$; it has three holes, two of type `term` and one of type `form`. Let's make the types explicit in our notation for syntax trees and write:

$$\begin{tikzcd}
 & & & x : \texttt{term}, y : \texttt{term}, a : \texttt{form} & \\
 & & & \texttt{conj} \arrow[dl] \arrow[dr] & \\
 & & \texttt{equal} \arrow[dl] \arrow[dr] & & \boxed{a} \\
 & \texttt{V}_{\texttt{f}} \arrow[dl] \arrow[dr] & & V_g \arrow[d] & \\
\boxed{x} & & \boxed{y} & \boxed{y} &
\end{tikzcd}$$

In this case, the types of the placeholders are redundant because the grammar forces us to plug in a term for x and y and a formula for a. It is still good to mark the types in the grammar: Firstly, it avoids confusion; secondly, the types are not always redundant.

Writing Expressions with Holes in LF LF lets us write expressions with holes explicitly. G from above is written as

$$[a][b](\texttt{impl}\ a\ (\wedge\ b\ a).$$

In LF, the holes of an expressions are always made explicit by prefixing the expressions with $[h]$ for every hole named h.

Again, we can argue about the notation. Some people prefer writing $G(a, b)$ or sometimes $G[a, b]$ if they want to emphasize that G has two holes a and b.

LF also lets us plug expressions into holes. If we want to plug in *true* for a and *false* for b in G, we write simply G *true false*. The result is the expression

$$G\ \mathit{true}\ \mathit{false}\ =\ \texttt{impl}\ \mathit{true}\ (\wedge\ \mathit{false}\ \mathit{true})$$

without holes.

Again, some people prefer the notation $G(\mathit{true}, \mathit{false})$; that is incidental. The bottom line is: We can write expressions with holes easily and concisely in LF.

LF also lets us make the types of placeholders explicit if we want to. For example, H from above is written with explicit types as

$$[x : \texttt{term}][y : \texttt{term}][a : \texttt{form}](\texttt{conj}\ (\texttt{equal}\ (V_f\ x\ y)\ y)\ a)$$

22.2.2 Expressions with Holes as Operations

Every expression E with a hole x can be seen as an operation: The input is some other expressions E'; the output is the result of plugging E' into the hole E. We can think of E as a program, E' as the input, and substitution as the machine that executes the program.

Since LF does substitution for us, we can directly use substitution to introduce some abbreviations. Consider the expression

$$[a : \texttt{form}][b : \texttt{form}](\texttt{conj}\ (\texttt{impl}\ a\ b)\ (\texttt{impl}\ b\ a)),$$

In LF, we can introduce a name for any expression, with or without holes. If we want to call the expression above `equiv`, we write

$$\texttt{equiv} : \texttt{form} \to \texttt{form} \to \texttt{form} \; = \; [a : \texttt{form}][b : \texttt{form}](\texttt{conj}\ (\texttt{impl}\ a\ b)\ (\texttt{impl}\ b\ a)).$$

We can now write

$$\texttt{equiv}\ \texttt{true}\ \texttt{false}$$

and substitution yields

$$\texttt{equiv}\ \texttt{true}\ \texttt{false} \; = \; \texttt{conj}\ (\texttt{impl}\ \texttt{true}\ \texttt{false})\ (\texttt{impl}\ \texttt{false}\ \texttt{true})$$

22.2.3 Productions using Expressions with Holes

We've seen already that holes in expressions are nothing special. We usually call them *free variables*, and they work in every grammar.

Using this intuition, we can now understand binding as an operation on expressions with holes.[1] For example, we can understand the universal quantifier $\forall$ as an operator that takes a single argument: a formula with a hole of type `term`.

Again the BNF notation for grammars does not provide a notation for productions that use expressions with holes. Therefore, we have to introduce one ourselves again. Consider two non-terminals A and B. Just like B represents the collection of all expressions derived from B, we write $\{x : A\}B$ for the collection of all expressions derived from B that have a hole named x of type A.

For example, consider the production

$$\texttt{form} ::= \texttt{conj}\ \#\ \texttt{form} \wedge \texttt{form}.$$

We can read it as

"The production `conj` takes two `form`s and produces a `form`."

Similarly, we read the new production

$$\texttt{form} ::= \texttt{univ}\ \#\ \{x : \texttt{term}\}\texttt{form}$$

as

"The production `univ` takes a `form` with a hole x for a `term` and produces a `form`."

With named productions and expressions with holes, our grammar for first-order logic becomes

$$\begin{array}{lcl}
\texttt{form} & ::= & \texttt{truth}\ \# \\
 & | & \texttt{falsity}\ \# \\
 & | & \texttt{conj}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{disj}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{impl}\ \#\ \texttt{form}, \texttt{form} \\
 & | & \texttt{neg}\ \#\ \texttt{form} \\
 & | & \texttt{univ}\ \#\ \{x : \texttt{term}\}\texttt{form} \\
 & | & \texttt{exist}\ \#\ \{x : \texttt{term}\}\texttt{form} \\
 & | & \texttt{equal}\ \#\ \texttt{term}, \texttt{term} \\
 & | & V_p\ \#\ \overbrace{\texttt{term}, \ldots, \texttt{term}}^{n} \text{ where } p \in \Sigma_p \text{ and } ar(p) = n \\
\texttt{term} & ::= & V_f\ \#\ \overbrace{\texttt{term}, \ldots, \texttt{term}}^{n} \text{ where } f \in \Sigma_f \text{ and } ar(f) = n \\
 & | & x
\end{array}$$

For example, let K be the formula $\forall x\ (p(x) \wedge \exists y\ q(u, y, v))$. Its syntax tree is

[1] This idea is called *higher-order abstract syntax*. It is one of the most important ideas in the area of logical frameworks and a central principle of LF.

$$\begin{tikzcd}
 & & u : \texttt{term}, v : \texttt{term} & & \\
 & & \texttt{univ} \arrow[d, "x : \texttt{term}"] & & \\
 & & \texttt{conj} \arrow[dl] \arrow[dr] & & \\
 & V_p \arrow[d] & & \texttt{exist} \arrow[d, "y : \texttt{term}"] & \\
 & \boxed{x} & & V_q \arrow[dl] \arrow[d] \arrow[dr] & \\
 & & \boxed{u} & \boxed{y} & \boxed{v}
\end{tikzcd}$$

Note how the nodes labelled with the productions `univ` and `exist` are followed by subtrees with holes. As before, the name of the hole is given above the respective subtree. Note that x and y are not holes of the overall tree; we say that they are *bound*. To distinguish them from holes of the whole tree, we use dashed boxes for the bound holes.

Productions with Holes in LF We can write down our new grammar for first-order logic directly in LF:

`form`	:	`type`.	
`term`	:	`type`.	
`truth`	:	`form`.	
`falsity`	:	`form`.	
`conj`	:	$\texttt{form} \to \texttt{form} \to \texttt{form}$.	
`disj`	:	$\texttt{form} \to \texttt{form} \to \texttt{form}$.	
`impl`	:	$\texttt{form} \to \texttt{form} \to \texttt{form}$.	
`neg`	:	$\texttt{form} \to \texttt{form}$.	
`equal`	:	$\texttt{term} \to \texttt{term} \to \texttt{form}$.	
`univ`	:	$(\{x : \texttt{term}\}\texttt{form}) \to \texttt{form}$.	
`exist`	:	$(\{x : \texttt{term}\}\texttt{form}) \to \texttt{form}$.	
V_p	:	$\overbrace{\texttt{term} \to \ldots \to \texttt{term}}^{n} \to \texttt{form}$.	where $p \in \Sigma_p$ and $ar(p) = n$
V_f	:	$\overbrace{\texttt{term} \to \ldots \to \texttt{term}}^{n} \to \texttt{term}$.	where $f \in \Sigma_f$ and $ar(f) = n$

The LF notation for syntax trees over these grammars is straightforward. For example, the formula K is written in LF as

$$[u : \texttt{term}][v : \texttt{term}]\ \texttt{univ}\ ([x : \texttt{term}]\ \texttt{conj}\ (p\ x)\ (\texttt{exist}\ ([y : \texttt{term}]\ V_q\ u\ y\ v)))$$

or – if we omit the types of the placeholders – as

$$[u][v]\ \texttt{univ}\ ([x]\ \texttt{conj}\ (p\ x)\ (\texttt{exist}\ ([y]\ V_q\ u\ y\ v)))$$

22.3 Notation Declarations

It is awkward to write syntax trees as

$$[u][v]\ \texttt{univ}\ ([x]\ \texttt{conj}\ (p\ x)\ (\texttt{exist}\ ([y]\ V_q\ u\ y\ v)))$$

because they are quite different from the nice notation

$$\forall x\ (p(x) \wedge \exists y\ q(u, y, v))$$

that we are used to. We can use a couple of syntactic sugars to tweak the LF notation into looking nicer.

Nice Identifiers We can give the productions any names we want. Most of the time, we can use the corresponding terminal symbol as the rule name. Moreover, we can use Unicode characters in identifiers. So we can change our grammar to

$$\begin{array}{llll}
\mathtt{form} & : & \mathtt{type}. & \\
\mathtt{term} & : & \mathtt{type}. & \\
\mathit{true} & : & \mathtt{form}. & \\
\mathit{false} & : & \mathtt{form}. & \\
\wedge & : & \mathtt{form} \to \mathtt{form} \to \mathtt{form}. & \\
\vee & : & \mathtt{form} \to \mathtt{form} \to \mathtt{form}. & \\
\Rightarrow & : & \mathtt{form} \to \mathtt{form} \to \mathtt{form}. & \\
\neg & : & \mathtt{form} \to \mathtt{form}. & \\
\leftrightarrow & : & \mathtt{form} \to \mathtt{form} \to \mathtt{form} & \\
 & = & [a:\mathtt{form}][b:\mathtt{form}](\wedge\ (\Rightarrow\ a\ b)\ (\Rightarrow\ b\ a)). & \\
\doteq & : & \mathtt{term} \to \mathtt{term} \to \mathtt{form}. & \\
\forall & : & (\{x:\mathtt{term}\}\mathtt{form}) \to \mathtt{form}. & \\
\exists & : & (\{x:\mathtt{term}\}\mathtt{form}) \to \mathtt{form}. & \\
p & : & \overbrace{\mathtt{term} \to \ldots \to \mathtt{term}}^{n} \to\ \mathtt{form}. & \text{where } p \in \Sigma_p \text{ and } ar(p) = n \\
f & : & \overbrace{\mathtt{term} \to \ldots \to \mathtt{term}}^{n} \to\ \mathtt{term}. & \text{where } f \in \Sigma_f \text{ and } ar(f) = n
\end{array}$$

Note that we include the abbreviation for equivalence from Sect. 22.2.2.
and now our example formula is written as

$$[u][v]\ \forall\ ([x]\ \ \wedge\ \ (p\ x)\ \exists\ ([y]\ q\ u\ y\ v)))$$

Notations We can also give notations to identifiers. A notation consists of

- fixity: prefix, infix (only for binary operators), or postfix
- if the fixity is infix, associativity: left, right, or none
  The associativity determines whether $P\ a\ Q\ a\ R$ is parsed as $(P\ a\ Q)\ a\ R$ (left), parsed as $P\ a\ (Q\ a\ R)$ (right), or forbidden (none). [2]
- precedence: a positive integer
  Higher precedences bind stronger, which permits omitting brackets.

We should make $\wedge$, $\vee$, $\Rightarrow$, $\leftrightarrow$, and $\doteq$ infix, and $\neg$ prefix. $\wedge$ and $\vee$ should be left- or right-associative so that we can write $P \wedge Q \wedge R$. $\Rightarrow$ and $\leftrightarrow$ should have no associativity to avoid ambiguity. $\neg$ should have a higher precedence than the other operators so that the brackets in $P \wedge (\neg Q)$ can be omitted. Finally, $\doteq$ should have a high precedence so that the brackets in $P \wedge (s \doteq t)$ can be omitted.
We can achieve that by adding the following notations to our grammar:

```
%infix left 10 $\wedge$.
%infix left 10 $\vee$.
%infix none 10 $\impl$.
%infix none 10 $\equiv$.
%prefix 15 $\neg$.
%infix none 20 $\doteq$.
```

Then our example expression becomes:

$$[u][v]\ \forall\ ([x]\ (p\ x) \wedge \exists\ ([y]\ q\ u\ y\ v)))$$

Finally, LF uses the convention that the $[-]$ operator has the lowest possible precedence. Therefore, brackets around $([x]F)$ can often be omitted. Our example expression finally becomes:

$$[u][v]\ \forall\ [x]\ (p\ x) \wedge \exists\ [y]\ (q\ u\ y\ v)$$

Note that this looks almost as nice as $\forall x\ (p(x) \wedge \exists y\ q(u, y, v))$. But the former is a syntax tree that we can work with directly whereas the latter is a string!

[2] Note that there is no such thing as an associative operator in LF.

22.4 The Fragment with Unary Dependent Types

Lesson 3: It's often useful to have infinite families of non-terminals. LF supports that elegantly.

Let's first understand why we would like to have infinitely many non-terminals.

22.4.1 Families of Non-Terminals

Proofs as Expressions So far we have seen how we can use LF to concisely write down context-sensitive grammars and expressions with holes. The main strength of LF is that we have a canonical linear notation for syntax trees of expressions.

In the area of logic, there is another kind of tree structure that is dominant: proofs. We have already seen how proofs can be represented as tree whose nodes are labelled with inference rules. Now we observe: This is basically the same principle as having syntax trees whose nodes are labelled with productions.

A production governs how we form new expressions from existing ones. Similarly, an inference rule governs how we form new proofs from existing ones. If we just think of them as syntax trees and productions, both of them become the same thing!

Compare a production, e.g., `impl`

$$\texttt{form} ::= \texttt{impl} \# \texttt{form}, \texttt{form}$$

with an inference rule, e.g., modus ponens

$$\frac{F \Rightarrow G \quad F}{G}\,mp$$

They look different, but that's just because we use very different notations. We see that if we write both of them as inference rules

$$\frac{\texttt{form} \quad \texttt{form}}{\texttt{form}}\,\texttt{impl} \qquad \frac{F \Rightarrow G \quad F}{G}\,mp$$

or both of them as productions

$$\texttt{form} ::= \texttt{impl} \# \texttt{form}, \texttt{form} \qquad G \ ::= \ mp \# F \Rightarrow G,\ F$$

Formulas as Non-Terminals However, there is one subtlety: In the case of inference rules, we do not have a finite set of non-terminal symbols. Indeed, the left side of the "production" for *mp* is G where G is an arbitrary formula. Everything would work if we had a way of getting a fresh non-terminal symbol for every formula. But because we have infinitely many formulas, this is not possible right away.

Therefore, LF makes a third (after named productions and expressions with holes) key addition to our understanding of grammars: infinite families of non-terminal symbols. More concretely, if $A : \texttt{type}$ is a non-terminal symbol, then we may add an A-indexed family B of non-terminal symbols as $B : A \to \texttt{type}$.

For example, we introduce a `form`-indexed family `proof` of non-terminal symbols as

$$\texttt{proof} : \texttt{form} \to \texttt{type}.$$

Then we have for every formula F a non-terminal symbol `proof` F. Expressions derived from `proof` F are the proofs of F.

Now we can in fact write our inference rule *mp* like this

$$\texttt{proof}\ G \ ::= \ mp \# \texttt{proof}\ (F \Rightarrow G),\ \texttt{proof}\ F$$

22.4.2 Dependent Type Constructors in LF

We have already seen the LF syntax for families of non-terminals above. For propositional logic, we declare our non-terminals as

$$\begin{array}{lcl} \texttt{form} & : & \texttt{type}. \\ \texttt{proof} & : & \texttt{form} \to \texttt{type}. \end{array}$$

We can think of `proof` as an operator that takes one argument and returns a non-terminal symbol. Then ordinary non-terminals like `form` are the special case of non-terminals with 0 arguments. In LF, we call the non-terminals *types*. Non-terminals like `proof` F are called *dependent types* because they depend on an expression.

The productions for `form` stay the same. And we want to write the production for mp as

"mp takes a proof of $F \Rightarrow G$ and a proof of F and produces a proof of G."

$$\frac{\vdash F \Rightarrow G \quad \vdash F}{\vdash G}\, mp$$

$$mp : \texttt{proof}\ (F \Rightarrow G) \to \texttt{proof}\ F \to \texttt{proof}\ G$$

But there is still a small problem: In all three of the above formulations of the rule, F and G occur out of nowhere. We kind of know that they are intended to be arbitrary formulas, but we're not precise about it. Therefore, let's say

"mp takes a formula F, a formula G, a proof of $F \Rightarrow G$, and a proof of F and produces a proof of G."

Now we notice something special that was hidden by the notation before: mp is a production that takes 4 arguments, but *the value of the first two arguments determines the type of the last two and of the return type*. This has become possible because of dependent types, and it is the most characteristic feature of dependent type theory.

Therefore, we finally write mp in LF as a production with two placeholders:

$$mp \ : \ \{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ (F \Rightarrow G) \to \texttt{proof}\ F \to \texttt{proof}\ G$$

Then our grammar for propositional logic looks like this:

`form`	:	`type`.	
`proof`	:	`form` $\to$ `type`.	
`truth`	:	`form`.	
`falsity`	:	`form`.	
`conj`	:	`form` $\to$ `form` $\to$ `form`.	
`disj`	:	`form` $\to$ `form` $\to$ `form`.	
`impl`	:	`form` $\to$ `form` $\to$ `form`.	
`neg`	:	`form` $\to$ `form`.	
V_p	:	`form`.	for $p \in \Sigma$
mp	:	$\{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ (F \Rightarrow G) \to \texttt{proof}\ F \to \texttt{proof}\ G$.	
$\wedge E_l$	:	$\{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ (F \wedge G) \to \texttt{proof}\ F$.	
$\wedge E_r$	:	$\{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ (F \wedge G) \to \texttt{proof}\ G$.	
$\wedge I$	:	$\{F : \texttt{form}\}\{G : \texttt{form}\}\,\texttt{proof}\ F \to \texttt{proof}\ G \to \texttt{proof}\ (F \wedge G)$.	

Here we have already included the rules for conjunction:

$$\frac{F \wedge G}{F}\,\wedge E_l \qquad \frac{F \wedge G}{G}\,\wedge E_r \qquad \frac{F \quad G}{F \wedge G}\,\wedge I$$

Of course, we're still missing all the other proof rules. We could write them down already as well, but it's better to make the LF background a tiny bit more formal first using Sect. 22.5.

Moreover, we can now declare axioms: An axiom asserting the formula F is declared as a symbol that returns a proof of F. That makes sense because axioms are like additional proofs that exist a priori.

For example, consider the theory of semigroups is the first-order signature with a binary function symbol and an axiom for associativity. We obtain it by adding declarations to first-order logic, one for the function and one for the axiom:

```
%sig Semigroup = {
  ∘      :  term → term → term.                        %infix left 30 ∘
  assoc  :  proof (∀[x]∀[y]∀[z](x ∘ y) ∘ z ≐ x ∘ (y ∘ z)).
}.
```

Here we make use of an additional feature of Twelf: We can give a list of declarations a name using the keywork `%sig`.

Note that we give $\circ$ a higher precedence than $\doteq$ to save brackets.

22.5 A Little More Background

22.5.1 Non-Terminals = Types

The basic idea behind the formal syntax of LF is that we have two sorts of objects: expressions and types. *Expressions* (sometimes also called *terms*) represent syntax and proof trees. *Types* classify the expressions; in particular, every non-terminal is a type. Every expression has exactly one type, and for an expression E of type T, we write $E : T : \texttt{type}$.

Thus, in order to understand the syntax of LF, we should understand what expressions and types there are. Let us review the different LF constructions that we have seen so far.

1. Plain expressions representing syntax/proof trees. Their type is given by the non-terminal from which they are derived. E.g.,

$$(\mathit{true} \vee p) \wedge q \ : \ \texttt{form} \ : \ \texttt{type}.$$

2. Expressions with holes

   - Expressions with one hole. Their type is the form $\{x : A\}B$ where A is the type of the hole x and B is the type of the whole expression. E.g.,

   $$[x : \texttt{term}](x \ \doteq \ x) \ : \ \{x : \texttt{term}\}\texttt{form} \ : \ \texttt{type}$$

     Note that a $[-]$-expression always has the corresponding $\{-\}$-type.

   - Expressions with multiple holes are a special case: They are formed by chaining the above construction. E.g.,

   $$[x : \texttt{term}][a : \texttt{form}](x \ \doteq \ x) \wedge a \ : \ \{x : \texttt{term}\}\{a : \texttt{form}\}\texttt{form} \ : \ \texttt{type}$$

   - Individual productions are expressions themselves. Their type is given in their declaration. E.g., $\neg : \texttt{form} \to \texttt{form}$ is one of our productions, and we obtain

   $$\neg \ : \ \texttt{form} \to \texttt{form} \ : \ \texttt{type}$$

     When we write down LF formally, this becomes just a special case of an expression with a hole: Indeed, we *define* $\texttt{form} \to \texttt{form}$ as an abbreviation of $\{a : \texttt{form}\}\texttt{form}$ (for some arbitrary name a), and we employ the *axiom*[3]

   $$\neg \ = \ [a : \texttt{form}]\neg a$$

     That makes sense because $\neg$ and $[a : \texttt{form}]\neg a$ behave in the same way according to $(*)$ below, namely

   $$\neg\, F \ = \ ([a : \texttt{form}]\neg a)\ F.$$

3. Expressions formed by application

[3] This is called η-conversion.

- Expressions formed by applying a production to another expression. Their type arises by removing one argument type. E.g., $\neg : \mathtt{form} \to \mathtt{form}$ is a production that takes one argument of type `form`, and if we apply it to another formula, we obtain

$$\neg\ ((\mathit{true} \vee p) \wedge q)\ :\ \mathtt{form}\ :\ \mathtt{type}$$

- Expressions formed by applying a production to multiple expressions are a special case: They are formed by chaining the above construction, e.g.,

$$\begin{array}{lcll} \mathit{mp} & : & \{F : \mathtt{form}\}\{G : \mathtt{form}\}\,\mathtt{proof}\ (F \Rightarrow G) \to \mathtt{proof}\ F \to \mathtt{proof}\ G & : \ \mathtt{type} \\ \mathit{mp\ true} & : & \{G : \mathtt{form}\}\,\mathtt{proof}\ (\mathit{true} \Rightarrow G) \to \mathtt{proof}\ \mathit{true} \to \mathtt{proof}\ G & : \ \mathtt{type} \\ \mathit{mp\ true\ p} & : & \mathtt{proof}\ (\mathit{true} \Rightarrow p) \to \mathtt{proof}\ \mathit{true} \to \mathtt{proof}\ p & : \ \mathtt{type} \end{array}$$

  and so on.

- Expressions formed by plugging an expression into a hole are a special case of application as well. Indeed, we have

$$[a : \mathtt{form}]\neg a\ :\ \{a : \mathtt{form}\}\mathtt{form}\ :\ \mathtt{type},$$

  and thus

$$([a : \mathtt{form}]\neg a)\ F\ :\ \mathtt{form}\ :\ \mathtt{type}$$

  for any formula F. Note that we already know the result of plugging F into the hole a, namely $\neg\ F$. Therefore, we employ the *axiom*[4]

$$([a : \mathtt{form}]\neg a)\ F\ =\ \neg\ F. \qquad (*)$$

Note that we only need four basic constructions to form all the LF expressions:

1. constants are atomic and must have been introduced by declarations; we distinguish the declarations
   (a) $a : \mathtt{type}$ and $b : A \to \mathtt{type}$ for non-terminals and families of non-terminals, respectively,
   (b) $c : A$ for some type A productions,
2. variables x represent placeholders, they are atomic and can be used exactly in the scope of an abstraction $[x : A]\mathit{scope}$, for some type A
3. abstractions represent terms with holes and are formed as $[x : A]t$,
4. applications represent production application and are formed as $t\ t'$.

For the types, it is even easier: We only need two basic constructors:

1. Atomic types arise from type declarations; we distinguish the
   (a) base types a, which are declared as $a : \mathtt{type}$,
   (b) dependent types $b\ t$, which are formed by applying a type family declared as $b : A \to \mathtt{type}$ to a term t of type A.
2. The type $\{x : A\}B$ of expressions of type B with a hole of type A (equivalently: the type of operations that take a term of type A and return a term of type B).

22.5.2 Types = Judgments

Now that we know the types of LF types a bit better, let's get back to our types `proof` F. There is something special about proofs: We often don't care about how many and what kinds of proofs of F, we have – the most important thing is to know whether there is *any* proof of F.[5] Consequently, we care about which of the following two is the case:

[4]This is called β-conversion.
[5]This attitude is called *proof irrelevance*.

1. There is no proof of F, i.e., there is no term of type `proof` F, i.e., the type `proof` F is empty.
2. There is some proof of F, i.e., there is a term of type `proof` F, i.e., the type `proof` F is non-empty.

It is very interesting to look at our types from that perspective.

Let's start with the atomic types. We read the non-emptiness of the type `proof` F as "F has a proof.". Thus, the type `proof` F expresses a judgment about F, namely the judgment

$$\begin{gathered}\texttt{proof } F \text{ is non-empty}\\ \text{means}\\ F \text{ is provable.}\end{gathered}$$

Secondly, let us look at the special case $A \to B$. (Recall: $A \to B$ is an abbreviation for $\{x : A\}B$ for some unused name x.) We notice the following: If the type $A \to B$ is non-empty (i.e., there is a term $f : A \to B$), then we have an operation that transforms terms of type A (i.e., terms $t : A$) into terms of type B (namely $f\ t : B$). Thus, we read

$$\begin{gathered}A \to B \text{ is non-empty}\\ \text{means}\\ \text{If } A \text{ is non-empty, then } B \text{ is non-empty.}\\ \text{or: } B \text{ is non-empty under the assumption that } A \text{ is non-empty.}\end{gathered}$$

For example, we read

$$\begin{gathered}\texttt{proof } F \to \texttt{proof } G \text{ is non-empty}\\ \text{means}\\ \text{If } F \text{ is provable, then } G \text{ is provable.}\\ \text{or: } G \text{ is provable under the assumption that } F \text{ is provable.}\end{gathered}$$

More concretely, the type `proof` $(F \wedge G) \to$ `proof` F (which is part of the rule $\wedge E_l$) expresses the judgment "If $F \wedge G$ is provable, then F is provable." We can also chain this construction, e.g., in the rule $\wedge I$, we write `proof` $F \to$ `proof` $G \to$ `proof`$(F \wedge G)$ to say "If F and G are provable, then $F \wedge G$ is provable".
Thirdly, let us look at the case $\{x : A\}B$ where x actually occurs in B. For example, consider the type $\{x : \texttt{form}\}\texttt{proof}\ x$. If $\{x : A\}B$ is non-empty, we have a term $t : \{x : A\}B$, i.e., a term with a hole of type A that yields a term of type $B\ t$ if we plug t into the hole. For example, if we have a term $\{x : \texttt{form}\}\texttt{proof}\ x$, then for any formula $F : \texttt{form}$, plugging F into the hole x in t yields a term of type `proof` F. Therefore, we read

$$\begin{gathered}\{x : A\}B \text{ is non-empty}\\ \text{means}\\ \text{For all } x : A, \text{ the type } B\ x \text{ is non-empty.}\\ \text{or: } B\ x \text{ is non-empty for arbitrary } x.\end{gathered}$$

For example, we read

$$\begin{gathered}\{x : \texttt{form}\}\texttt{proof}\ x \text{ is non-empty}\\ \text{means}\\ \text{For all formulas } x,\ x \text{ is provable.}\\ \text{or: } x \text{ is provable for arbitrary } x.\end{gathered}$$

Obviously, the judgment "For all formulas x, x is provable." is should not hold. The type $\{x : \texttt{form}\}\texttt{proof}\ x$ represents the judgment that our logic has a contradiction and should always be empty. We formalize this intuition in LF by adding the definition

$$\lightning : \texttt{type} = \{x : \texttt{form}\}\texttt{proof}\ x.$$

22.6 Collecting the Pieces: A Logical Framework

We are now ready to use LF to write down the remaining proof rules. The key trick is the intuition non-terminals, types, and judgments are all the same thing.

We will start with the rule for implication, negation, and universal quantification, which have a very nice property: They establish a back-and-forth mapping between the respective type `proof` F and some other type. More concretely, we have the following correspondence:

proof type	represented as	intuition
$\texttt{proof}\ (F \Rightarrow G)$	$\texttt{proof}\ F \rightarrow \texttt{proof}\ G$	if F, then G
$\texttt{proof}\ (\neg F)$	$\texttt{proof}\ F \rightarrow \lightning$	if F, then contradiction
$\texttt{proof}\ (\forall[x : \texttt{term}](F\ x))$	$\{x : \texttt{term}\}\texttt{proof}\ (F\ x)$	for arbitrary $x : \texttt{term}$, $F\ x$

The introduction rules go from the representing type to the proof type. The elimination rules go the other way round.

Implication Let's start with the implication introduction rule $\Rightarrow I$, which we can express in any one of the following ways:

$\Rightarrow I$ takes two formulas F, G, a proof of G with a hole for a proof of F and produces a proof of $F \Rightarrow G$.

For arbitrary F, G, if G is provable under the assumption that F is provable, then $F \Rightarrow G$ is provable.

$$\frac{F : \texttt{form} \quad G : \texttt{form} \quad F \vdash G}{\vdash F \Rightarrow G} \Rightarrow I$$

$$\Rightarrow I : \{F : \texttt{form}\}\{G : \texttt{form}\}(\texttt{proof}\ F \rightarrow \texttt{proof}\ G) \rightarrow \texttt{proof}\ (F \Rightarrow G)$$

Note how we are careful to introduce the arbitrary parameters F and G. This is good practice. But it can get quite tedious to do that, and we will omit them later, see Sect. 22.8.

We have already represented the implication elimination rule – modus ponens:

$\Rightarrow E$ takes two formulas F, G, a proof of $F \Rightarrow G$, and a proof of F and produces a proof of G.

For arbitrary F, G, if $F \Rightarrow G$ and F are provable, then G is provable.

$$\frac{F : \texttt{form} \quad G : \texttt{form} \quad \vdash F \Rightarrow G \quad \vdash F}{\vdash G} \Rightarrow E$$

$$\Rightarrow E : \{F : \texttt{form}\}\{G : \texttt{form}\}\texttt{proof}\ (F \Rightarrow G) \rightarrow \texttt{proof}\ F \rightarrow \texttt{proof}\ G.$$

Negation The rule for negation introduction is similar to that of implication introduction. Intuitively, we already know it as proof by contradiction:

$\neg I$ takes a formula F and a proof of contraction with a hole for a proof of F
and produces a proof of $\neg F$.

For arbitrary F, if a contradiction is provable under the assumption that F is provable,
then $\neg F$ is provable.

$$\frac{F : \texttt{form} \quad F \vdash \lightning}{\vdash \neg F} \neg I$$

$$\neg I : \{F : \texttt{form}\}(\texttt{proof}\ F \rightarrow \lightning) \rightarrow \texttt{proof}\ (\neg F)$$

The elimination rule for negation is straightforward – we know it as contradiction:

$\neg E$ takes a formula F, two proofs of $\neg F$ and F and produces a contradiction.

For arbitrary F, if both $\neg F$ and F are provable, then we have a contradiction.

$$\frac{F : \mathtt{form} \quad \vdash \neg F \quad \vdash F}{\vdash \lightning}\neg E$$

$$\neg E : \{F : \mathtt{form}\}\mathtt{proof}\ (\neg F) \to \mathtt{proof}\ F \to \lightning.$$

Universal Quantification Next we look at the rules for universally quantified formulas $\forall[x : \mathtt{term}](F\ x)$. Note that, here, F is an arbitrary formula with a hole of type `term`. The introduction rule $\forall I$ can be expressed as follows:

$\forall I$ takes a formula F with a `term`-hole, and a proof of $F\ x$ with a `term`-hole x and produces a proof of $\forall[x : \mathtt{term}](F\ x)$.

For an arbitrary formula F with a `term`-hole, if $F\ x$ is provable for arbitrary $x : \mathtt{term}$, then $\forall[x : \mathtt{term}](F\ x)$ is provable.

$$\frac{F : \{x : \mathtt{term}\}\mathtt{form} \quad x : \mathtt{term} \vdash F\ x}{\vdash \forall[x : \mathtt{term}]F\ x}\Rightarrow I$$

$$\Rightarrow I : \{F : \{x : \mathtt{term}\}\mathtt{form}\}\ \big(\{x : \mathtt{term}\}\mathtt{proof}\ (F\ x)\big) \to \mathtt{proof}\ (\forall[x : \mathtt{term}](F\ x))$$

The corresponding elimination rule is straightforward – we know it as instantiation:

$\forall E$ takes a formula F with a `term`-hole, a proof of $\forall[x : \mathtt{term}](F\ x)$, and a term x and produces a proof of $F\ x$.

For an arbitrary formula F with a `term`-hole, if $\forall[x : \mathtt{term}](F\ x)$ is provable, then $F\ x$ is provable for any term x.

$$\frac{F : \{x : \mathtt{term}\}\mathtt{form} \quad \vdash \forall[x : \mathtt{term}]F\ x \quad x : \mathtt{term}}{\vdash F\ x}\Rightarrow I$$

$$\Rightarrow I : \{F : \{x : \mathtt{term}\}\mathtt{form}\}\ \mathtt{proof}\ (\forall[x : \mathtt{term}](F\ x)) \to \{x : \mathtt{term}\}\mathtt{proof}\ (F\ x)$$

Rules Remaining The six remaining rules are introduction and elimination for conjunction, disjunction, and existential quantification. They cannot be represented in the same nice way using a representing type. However, their representation is still very natural.

Introduction of conjunction and existential. These two behave similarly to pairing: To prove $F \wedge G$, we need a proof of F *and* a proof of G. To prove $\exists[x : \mathtt{term}](F\ x)$, we need a term t *and* a proof of $F\ t$. Consequently, the rules are

$$\wedge I : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\mathtt{proof}\ F \to \mathtt{proof}\ G \to \mathtt{proof}\ (F \wedge G)$$

$$\exists I : \{F : \{x : \mathtt{term}\}\mathtt{form}\}\{t : \mathtt{term}\}\mathtt{proof}\ (F\ t) \to \mathtt{proof}\ (\exists[x : \mathtt{term}](F\ x))$$

Elimination of conjunction and introduction of disjunction. Here we need two pairs of rules – one for the left and one for the right subformula. The resulting rules are dual:

$$\wedge E_l : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\mathtt{proof}\ (F \wedge G) \to \mathtt{proof}\ F$$
$$\wedge E_r : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\mathtt{proof}\ (F \wedge G) \to \mathtt{proof}\ G$$

$$\vee I_l : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\mathtt{proof}\ F \to \mathtt{proof}(F \vee G)$$
$$\vee I_r : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\mathtt{proof}\ G \to \mathtt{proof}(F \vee G)$$

Elimination of disjunction and existential. These are the most difficult rules because there is an inherent indeterminism: If we have a proof of $F \vee G$, we know one of the two holds, but we do not know which one; similarly, if $\exists[x : \mathtt{term}](F\ x)$ holds, we know $F\ t$ holds for some t, but we do not know for which t. Therefore, we need to employ case distinction.

Let C be the formula be the formula, we want to prove. If we want to use $F \vee G$ in the proof, we need to distinguish two cases:

$$\frac{F : \mathtt{form} \quad G : \mathtt{form} \quad C : \mathtt{form} \quad \vdash F \vee G \quad F \vdash C \quad G \vdash C}{\vdash C}\vee E$$

$$\begin{gathered}\vee E : \{F : \mathtt{form}\}\{G : \mathtt{form}\}\{C : \mathtt{form}\}\\ \mathtt{proof}\ (F \vee G) \to (\mathtt{proof}\ F \to \mathtt{proof}\ C) \to (\mathtt{proof}\ G \to \mathtt{proof}\ C) \to \mathtt{proof}\ C.\end{gathered}$$

Similarly, if we want to use $\exists[x : \mathtt{term}](F\ x)$ in the proof of C, we need to distinguish one case for every possible t. We can merge all those cases into one proof with a hole for t:

$$\frac{F : \{x : \mathtt{term}\}\mathtt{form} \quad C : \mathtt{form} \quad \vdash \exists[x : \mathtt{term}](F\ x) \quad t : \mathtt{term},\ F\ t \vdash C}{\vdash C}\exists E$$

$$\begin{gathered}\exists E : \{F : \{x : \mathtt{term}\}\mathtt{form}\}\{C : \mathtt{form}\}\\ \mathtt{proof}\ \big(\exists[x : \mathtt{term}](F\ x)\big) \to \big(\{t : \mathtt{term}\}\mathtt{proof}\ (F\ t) \to \mathtt{proof}\ C\big) \to \mathtt{proof}\ C.\end{gathered}$$

22.7 Proofs as Expressions

We can now write natural deduction proofs directly as LF expressions. This is important because we do not have any other linear way to write formal proofs. The only alternative is the proof tree notation, which is elegant to read but awkward to write.

Consider the natural deduction proof of $(A \wedge B) \Rightarrow (B \wedge A)$. It looks like this

$$\dfrac{\vdash (A \wedge B) \Rightarrow (B \wedge A)}{\dfrac{A \wedge B \vdash B \wedge A}{\dfrac{A \wedge B \vdash B}{\dfrac{A \wedge B \vdash A \wedge B}{}\,Axiom}\,\wedge E_r \qquad \dfrac{A \wedge B \vdash A}{\dfrac{A \wedge B \vdash A \wedge B}{}\,Axiom}\,\wedge E_l}\,\wedge I}\,\Rightarrow I$$

Let's write this in a slightly more precise way by adding all the parameters of the proof rules, which we usually leave implicit:

$$\dfrac{\vdash (A \wedge B) \Rightarrow (B \wedge A)}{\dfrac{A \wedge B \vdash B \wedge A}{\dfrac{A \wedge B \vdash B}{\dfrac{A \wedge B \vdash A \wedge B}{}\,Axiom\ A \wedge B}\,\wedge E_r\ A\ B \qquad \dfrac{A \wedge B \vdash A}{\dfrac{A \wedge B \vdash A \wedge B}{}\,Axiom\ A \wedge B}\,\wedge E_l\ A\ B}\,\wedge I\ B\ A}\,\Rightarrow I\ (A \wedge B)\ (B \wedge A)$$

From such a proof, we can read off the corresponding LF term right away:

$$\Rightarrow I\ (A \wedge B)\ (B \wedge A)\ \big([p : \mathtt{proof}(A \wedge B)]\ \wedge I\ B\ A\ (\wedge E_l\ A\ B\ p)\ (\wedge E_r\ A\ B\ p)\big)$$

Note how the $\Rightarrow I$ rule, which introduces an assumption on the left of the $\vdash$, is represented as a term with a hole p. Moreover, whenever we refer to an assumption using the *Axiom* rule, we simply give the name of hole. (In particular, we do not have to represent the *Axiom* explicitly in LF – we get automatically from the terms-with-holes formalism.)

We can add such a theorem to our formalism by using an abbreviation:

$$\begin{aligned}\wedge comm \quad &: \quad \{A : \mathtt{form}\}\{B : \mathtt{form}\}\ \mathtt{proof}\ \big((A \wedge B) \Rightarrow (B \wedge A)\big)\\ &= \quad [A : \mathtt{form}][B : \mathtt{form}]\ \Rightarrow I\ (A \wedge B)\ (B \wedge A)\ \big([p : \mathtt{proof}(A \wedge B)]\ \wedge I\ B\ A\ (\wedge E_l\ A\ B\ p)\ (\wedge E_r\ A\ B\ p)\big).\end{aligned}$$

22.8 Implicit Arguments

We have already remarked above that we often do not mention some of the parameters in proof rules. For example, we prefer writing

$$\frac{F \vdash G}{\vdash F \Rightarrow G} \Rightarrow I$$

to

$$\frac{F : \texttt{form} \quad G : \texttt{form} \quad F \vdash G}{\vdash F \Rightarrow G} \Rightarrow I$$

Twelf supports that: We can write

$$\Rightarrow I : (\texttt{proof}\ F \to \texttt{proof}\ G) \to \texttt{proof}\ (F \Rightarrow G)$$

and Twelf will implicitly take it to mean

$$\Rightarrow I : \{F : \texttt{form}\}\{G : \texttt{form}\}(\texttt{proof}\ F \to \texttt{proof}\ G) \to \texttt{proof}\ (F \Rightarrow G)$$

More precisely: Whenever Twelf finds a free variable F in a declaration, it will figure out the type T of F and place $\{F : T\}$ in front of the type. F is called an *implicit argument*.

Twelf doing things automatically also means that sometimes errors go unnoticed. Therefore, Twelf uses the following convention: Variable names for implicit arguments must start with an upper case letter. Other free variables simply yield errors.

Above, we have only said that proof rules do not have to declare parameters explicitly. The real power of Twelf is that when using a proof rule, we also do not have to provide the concrete values for these parameters – Twelf figures them out based on the context.

For example, in the proof term given in Sect. 22.7, we can omit all the gray parts.

Chapter 23

Syntax of LF

We will now look at the logical framework LF ([?]) more formally. If you are already familiar with the Twelf implementation [?] of LF, consider Fig. 26.1 for the correspondence between the mathematical notation of LF and the ASCII notation of Twelf. LF is a type theory that arises from STT by adding the feature of **dependent types**. Dependent typing means that terms may occur in types. We will use that to represent proofs as terms and judgment as types: A constant declaration $\mathtt{proof} : \mathtt{form} \to \mathtt{type}$ returns a new type for every formula – the type of proofs of that formula.

In dependent type theories, there are several type constructors. The most important one is the dependent function type. In a dependent function type, the value of the argument may occur in the return type. More generally, using functions with multiple arguments, the value of argument may occur in the later argument types and the return type. For example,

$$\Pi F : \mathtt{form}.\Pi G : \mathtt{form}.\mathtt{proof}\ F \to\ \mathtt{proof}\ G\ \to \mathtt{proof}\ (F \wedge G)$$

is the type of the conjunction introduction rule as a function taking two formulas and two proofs of them and returning a proof of the conjunction.

A good way to understand the syntax of LF is to compare it to the syntax of STT. The **judgments** of LF are almost the same as for STT. There are only two differences between STT and LF:

$\vdash \Sigma$	Σ is a well-formed signature
$\vdash \sigma : \Sigma \to \Sigma'$	σ is a well-formed signature morphism from Σ to Σ'
$\vdash_\Sigma \Gamma$	Γ is a well-formed Σ-context
$\vdash_\Sigma \gamma : \Gamma \to \Gamma'$	γ is a well-formed substitution from Γ to Γ' over Σ
$\Gamma \vdash_\Sigma K : \mathtt{kind}$	K is a well-formed kind over Σ and Γ
$\Gamma \vdash_\Sigma A\ :\ K$	A is a well-formed type family of (well-formed) kind K over Σ and Γ
$\Gamma \vdash_\Sigma t\ :\ A$	t is a well-formed term of (well-formed) type A over Σ and Γ

The first difference is that LF adds a judgment for well-formed **kinds**. In type theories, the expressions are arranged in a hierarchy where the colon judgment $E : F$ is used if F is one level above E in the hierarchy. The first three levels have special names: level 0 is the term level, level 1 the type level, and level 2 the kind level. We might call level 3 the hyperkind level.

The kind level has the special kind `type`, which is the kind of all types. There may be more type-level expressions, which are then kinded by other kinds. Similarly, `kind` is the hyperkind of all kinds. If $E : F$ holds for a term E, then F must be a type, i.e., $E : F : \mathtt{type}$. Similarly, if E is a type-level expression, then F must be a kind and thus $E : F : \mathtt{kind}$. We say terms are typed by types, and type-level expressions are kinded by kinds.

In STT, `type` is the only kind, and all type-level expressions are types. Thus, kinds can be ignored. LF has more than one kind, and therefore, we need an extra judgment for well-formed kinds. The additional type level expressions are called type families. Thus, LF has typed terms and kinded type families.

The second difference is the judgment for well-formed type families. It now depends on a context Γ because terms (and thus variables) can occur in types.

The context-free **grammar** also looks very similar:

$$\begin{array}{lll}
\Sigma & ::= & \cdot \,|\, \Sigma,\ c : A \,|\, \Sigma,\ a : K \\
\sigma & ::= & \cdot \,|\, \sigma,\ c := t \,|\, \sigma,\ a := A \\
\Gamma & ::= & \cdot \,|\, \Gamma,\ x : A \\
\gamma & ::= & \cdot \,|\, \gamma,\ x/t \\
K & ::= & \mathtt{type} \,|\, A \to K \\
A & ::= & a \,|\, A\ t \,|\, \Pi x : A.A \\
t & ::= & c \,|\, x \,|\, t\ t \,|\, \lambda x : A.t
\end{array}$$

The first difference between the grammars is that signatures may contain type family declarations $a : K$, not just base type declaration $a : \mathtt{type}$. This is a natural consequence of using kinds.

The second difference is the extra non-terminal K for kinds. A kind is either the special kind type or of the form $A \to K$. It is easy to see that all kinds are of the form $A_1 \to \ldots \to A_n \to \mathtt{type}$. The intuition of such a kind is that a type family constant $a : A_1 \to \ldots \to A_n \to \mathtt{type}$ takes n arguments of types $A_1, \ldots, A_n$ and returns a type. This also means that other imaginable kinds are excluded: For example, there is no kind $\mathtt{type} \to \mathtt{type}$ as we would need for a type operator like *list*.

The third difference are the productions for type families. The $\to$ is gone, instead we have a Π, and we add an application $A\ t$ of type families to terms. The application is easy to understand: If we have $A : B \to \mathtt{type}$ and $t : B$, then we need a way to apply A to t in order to get a type – that application is $A\ t$. The Π is the dependent function type constructor. Intuitively, $\Pi x : A.B$ is the type of functions that take an argument x of type A and return a value of type $B(x)$ that depends on the value of the argument. Types of functions with multiple arguments are formed by chaining Π just like chaining $\to$ in STT: $\Pi x_1 : A_1. \ldots \Pi x_n : A_n.B$ is the type of functions taking n arguments x_i of type A_i and returning a value of type B where every argument x_i may occur in all later argument types $A_{i+1}, \ldots, A_n$ and the return type B. The $\to$ of STT is recovered as a special case: If in $\Pi x : A.B$, x does not occur in B, we abbreviate it as $A \to B$.

Intuitively, we have:

- Signatures are lists of typed constants ($c : A$) and kinded type families ($a : K$).
- Signature morphisms are lists of terms and type families to be substituted for the symbols of a signature.
- Contexts are lists of typed variables ($x : A$). (As in STT, there are no kinded variables.)
- Substitutions are lists of terms to be substituted for the variables (x/t).
- Kinds are of the form $A_1 \to \ldots \to A_n \to \mathtt{type}$ (just writing type if $n = 0$).
- Types are base types (a), application of type families to terms ($A\ t$) or dependent function types ($\Pi x : A.B$).
- Terms are constants (c), variables (x), dependent function applications ($f\ t$), or dependent λ-abstractions ($\lambda x : A.t$).

Example 23.1. The following signature represents some operations on matrices.

- A type for natural numbers with 1 and successor: $N : \mathtt{type}$, $1 : N$, $s : N \to N$. Here $s : N \to N$ abbreviates $s : \Pi x : N.N$.
- A type family for matrices: $Mat : N \to N \to \mathtt{type}$. Mat takes two natural numbers, say m and n, and returns a new type – the type of (m, n)-matrices (over same ring which we omit).
- A constant for addition of matrices:

  $$+ : \Pi x : N.\Pi y : N.Mat\ x\ y \to Mat\ x\ y \to Mat\ x\ y.$$

  $+$ first takes two natural numbers x and y and then two matrices with the dimensions (x, y), i.e., two elements of type $Mat\ x\ y$, and returns another such matrix.
- A constant for multiplication of matrices:

  $$\cdot : \Pi x : N.\Pi y : N.\Pi z : N.Mat\ x\ y \to Mat\ y\ z \to Mat\ x\ z.$$

  $\cdot$ first takes three natural numbers x, y, and z and then two matrices with the dimensions (x, y) and (y, z) and returns a matrix of dimension (x, z).

Here is the declaration of $+$ again without the $\to$ abbreviation and with some more brackets:

$$+ \quad : \quad \Pi x : N.\big(\Pi y : N.\big(\Pi m : (Mat\ x)\ y.\big(\Pi n : (Mat\ x)\ y.\big((Mat\ x)\ y\big)\big)\big)\big)$$

The inference system for these judgments is more complicated than the one for STT because all judgments (except for signature morphisms) have to be defined by mutual recursion. This is because terms occur in types, types occur in kinds, and types and kinds occur in contexts and signatures – on the other hand well-formed terms, type families, and kinds are defined relative to a signature and a context. It is even worse: Even substitution application must be part of that big mutual recursion because it is needed already in some of the rules.

This big mutual recursion can be hard to grasp for a beginner. Apart from that, it is quite straightforward. The rules are given in Fig. 23.1 and 23.2. Some expressions are underlined as a visual aid to make the bracketing structure apparent.

$$\frac{}{\vdash \cdot}\ sigempty \qquad \frac{\vdash \Sigma \quad c \text{ not in } \Sigma \quad \cdot \vdash_\Sigma A\ :\ \mathtt{type}}{\vdash \Sigma,\ c : A}\ sigcon \qquad \frac{\vdash \Sigma \quad a \text{ not in } \Sigma \quad \cdot \vdash_\Sigma K : \mathtt{kind}}{\vdash \Sigma,\ a : K}\ sigtype$$

$$\frac{\vdash \Sigma'}{\vdash \cdot : \cdot \to \Sigma'}\ morphempty \qquad \frac{\vdash \sigma : \Sigma \to \Sigma' \quad \cdot \vdash_{\Sigma'} t\ :\ \overline{\sigma}(A)}{\vdash \underline{\sigma,\ c := t} : \underline{\Sigma,\ c : A} \to \Sigma'}\ morphcon \qquad \frac{\vdash \sigma : \Sigma \to \Sigma' \quad \cdot \vdash_\Sigma A\ :\ \overline{\sigma}(K)}{\vdash \underline{\sigma,\ a := A} : \underline{\Sigma,\ a : K} \to \Sigma'}\ morphtype$$

$$\frac{\vdash \Sigma}{\vdash_\Sigma \cdot}\ conempty \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A\ :\ \mathtt{type}}{\vdash_\Sigma \Gamma,\ x : A}\ convar$$

$$\frac{\vdash_\Sigma \Gamma'\ \checkmark}{\vdash_\Sigma \cdot : \cdot \to \Gamma'}\ subsempty \qquad \frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma' \vdash_\Sigma t\ :\ \overline{\gamma}(A)}{\vdash_\Sigma \underline{\gamma,\ x/t} : \underline{\Gamma,\ x : A} \to \Gamma'}\ subsvar$$

Figure 23.1: Well-formed Signatures, Morphisms, Contexts, and Substitutions

Note that the types and kinds of the constants in signatures must be closed, i.e., be well-formed in the empty context. In a context, however, the type of a variable may refer to the preceding variables. Therefore, when defining substitutions, we have to already apply the substitution to the types of the domain context in rule *subsvar*.

The rules for terms almost look like the ones from STT, but note that variables may now occur in the types. Therefore, substitution application is needed in rule *termapp*.

Definition 23.2 (Substitution Application)**.** For a substitution $\gamma = x_1/t_1, \ldots, x_n/t_n$, and a term, type family, or kind using only the variables $x_1, \ldots, x_n$, we define $\overline{\gamma}(-)$ as follows:

- $\overline{\gamma}(\mathtt{type}) = \mathtt{type}$,
- $\overline{\gamma}(A \to K) = \overline{\gamma}(A) \to \gamma(K)$,
- $\overline{\gamma}(a) = a$,
- $\overline{\gamma}(A\ t) = \overline{\gamma}(A)\ \overline{\gamma}(t)$,
- $\overline{\gamma}(\Pi x : A.t) = \Pi x : \overline{\gamma}(A).\overline{\gamma^x}(t)$,
- $\overline{\gamma}(c) = c$,
- $\overline{\gamma}(x_i) = t_i$,
- $\overline{\gamma}(f\ t) = \overline{\gamma}(f)\ \overline{\gamma}(t)$,
- $\overline{\gamma}(\lambda x : A.t) = \lambda x : \overline{\gamma}(A).\overline{\gamma^x}(t)$,

where γ^x abbreviates $\gamma, x/x$.

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathtt{type} : \mathtt{kind}}\,kdbase \qquad \frac{\Gamma \vdash_\Sigma A \;:\; \mathtt{type} \quad \Gamma \vdash_\Sigma K : \mathtt{kind}}{\Gamma \vdash_\Sigma A \to K : \mathtt{kind}}\,kdfun$$

$$\frac{\vdash_\Sigma \Gamma \quad a : K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a \;:\; K}\,tpbase \qquad \frac{\Gamma \vdash_\Sigma C \;:\; A \to K \quad \Gamma \vdash_\Sigma t \;:\; A}{\Gamma \vdash_\Sigma C\, t \;:\; K}\,tpapp$$

$$\frac{\Gamma \vdash_\Sigma A \;:\; \mathtt{type} \quad \Gamma, x : A \vdash_\Sigma B \;:\; \mathtt{type}}{\Gamma \vdash_\Sigma \Pi x : A.B \;:\; \mathtt{type}}\,tpfun$$

$$\frac{c : A \text{ in } \Sigma \quad \vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma c \;:\; A}\,termcon \qquad \frac{x : A \text{ in } \Gamma \text{ (rightmost } x \text{ if multiple)} \quad \vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma x \;:\; A}\,termvar$$

$$\frac{\Gamma \vdash_\Sigma f \;:\; \Pi x : A.B \quad \Gamma \vdash_\Sigma t \;:\; A}{\Gamma \vdash_\Sigma f\; t \;:\; B[x/t]}\,termapp \qquad \frac{\Gamma,\; x : A \vdash_\Sigma t \;:\; B}{\Gamma \vdash_\Sigma \lambda x : A.t \;:\; \Pi x : A.B}\,termlam$$

Figure 23.2: Well-formed Kinds, Type Families, and Kinds

We also write $t[x/s]$ for the result of substituting x with s and all other variables with themselves.

Lemma 23.3 (Substitution Application)**.** *Assume* $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$. *Then*

$$\Gamma \vdash_\Sigma t \;:\; A \quad \text{implies} \quad \Gamma' \vdash_\Sigma \overline{\gamma}(t) \;:\; \overline{\gamma}(A)$$

$$\Gamma \vdash_\Sigma A \;:\; K \quad \text{implies} \quad \Gamma' \vdash_\Sigma \overline{\gamma}(A) \;:\; \overline{\gamma}(K)$$

$$\Gamma \vdash_\Sigma K : \mathtt{kind} \quad \text{implies} \quad \Gamma' \vdash_\Sigma \overline{\gamma}(K) : \mathtt{kind}$$

Proof. One big induction. □

Similarly, we define signature morphism application.

Definition 23.4 (Morphism Application)**.** Assume a signature morphism $\vdash \sigma : \Sigma \to \Sigma'$. Then, for a Σ term, type family, or kind we define $\overline{\sigma}(-)$ as follows:

- $\overline{\sigma}(\mathtt{type}) = \mathtt{type}$,
- $\overline{\sigma}(A \to K) = \overline{\sigma}(A) \to \sigma(K)$,
- $\overline{\sigma}(a) = A$ where $a := A$ in σ,
- $\overline{\sigma}(A\; t) = \overline{\sigma}(A)\; \overline{\sigma}(t)$,
- $\overline{\sigma}(\Pi x : A.B) = \Pi x : \overline{\sigma}(A).\overline{\sigma}(B)$,
- $\overline{\sigma}(c) = t$ where $c := t$ in σ,
- $\overline{\sigma}(x) = x$,
- $\overline{\sigma}(f\; t) = \overline{\sigma}(f)\; \overline{\sigma}(t)$,
- $\overline{\sigma}(\lambda x : A.t) = \lambda x : \overline{\sigma}(A).\overline{\sigma}(t)$.

Like for STT, we only care about applying morphisms to closed expressions and map (bound) variables to themselves.

Lemma 23.5 (Morphism Application)**.** *Assume* $\vdash \sigma : \Sigma \to \Sigma'$*. Then*

$$\cdot \vdash_\Sigma t \ : \ A \qquad \text{implies} \qquad \cdot \vdash_{\Sigma'} \overline{\sigma}(t) \ : \ \overline{\sigma}(A)$$

$$\cdot \vdash_\Sigma A \ : \ K \qquad \text{implies} \qquad \cdot \vdash_\Sigma \overline{\sigma}(A) \ : \ \overline{\sigma}(K)$$

$$\cdot \vdash_\Sigma K : \texttt{kind} \qquad \text{implies} \qquad \cdot \vdash_\Sigma \overline{\sigma}(K) : \texttt{kind}$$

Proof. Another big induction.
We actually use a more general theorem as the induction hypothesis:

$$\Gamma \vdash_\Sigma t \ : \ A \qquad \text{implies} \qquad \overline{\sigma}(\Gamma) \vdash_{\Sigma'} \overline{\sigma}(t) \ : \ \overline{\sigma}(A)$$

$$\Gamma \vdash_\Sigma A \ : \ K \qquad \text{implies} \qquad \overline{\sigma}(\Gamma) \vdash_\Sigma \overline{\sigma}(A) \ : \ \overline{\sigma}(K)$$

$$\Gamma \vdash_\Sigma K : \texttt{kind} \qquad \text{implies} \qquad \overline{\sigma}(\Gamma) \vdash_\Sigma \overline{\sigma}(K) : \texttt{kind}$$

$$\vdash_\Sigma \Gamma \checkmark \qquad \text{implies} \qquad \vdash_\Sigma \overline{\sigma}(\Gamma) \checkmark$$

Here we define $\overline{\sigma}(\Gamma)$ (the homomorphic translation of a context along a signature morphism) by

$$\overline{\sigma}(\cdot) = \cdot$$

$$\overline{\sigma}(\Gamma,\, x : A) = \overline{\sigma}(\Gamma),\, x : \overline{\sigma}(A)$$

□

Chapter 24

Semantics of LF

LF does not have formulas or a consequence relation between them. Therefore, LF is not a logic. Nevertheless, LF is a type theory, and every type theory has a semantics. As for logics, there is a proof- and a model-theoretical approach to define the semantics.

24.1 Model-Theoretic Semantics

The model theory of LF can be defined very similarly to the model theory of STT.

Just like the syntax of LF needs one higher type-level than STT (namely for kinds), the model theory of LF requires one higher set-theoretical concept, namely classes. Intuitively, any collection of sets is a class. For example, $\mathcal{SET}$ is the class containing all sets; $\varnothing$ is the class containing no set; every set is a class. Some classes are so "big" that they are not sets anymore, e.g., $\mathcal{SET}$ is not a set; other classes are so "small" that they are sets, e.g., $\varnothing$ is a set.

Then we can interpret `kind` as $\mathcal{CLASS}$ and thus kinds as classes; `type` as $\mathcal{SET}$ and thus types as sets; and terms as elements of sets. We have:

Expression	Syntax	Semantics
kind K	$\Gamma \vdash_\Sigma K : \texttt{kind}$	$[\![K]\!]^{I,\alpha} \in [\![\texttt{kind}]\!] = \mathcal{CLASS}$
type family A	$\Gamma \vdash_\Sigma A \ : \ K$	$[\![\Gamma\|A]\!]^{I,\alpha} \in [\![\Gamma\|K]\!]^{I,\alpha}$
in particular: type A	$\Gamma \vdash_\Sigma A \ : \ \texttt{type}$	$[\![\Gamma\|A]\!]^{I,\alpha} \in [\![\Gamma\|\texttt{type}]\!]^{I,\alpha} = \mathcal{SET}$
term t	$\Gamma \vdash_\Sigma t \ : \ A$	$[\![\Gamma\|t]\!]^{I,\alpha} \in [\![\Gamma\|A]\!]^{I,\alpha}$

Because the syntax of LF is defined in by mutual recursion, the same must be the case for the semantics: Models, assignments, and the interpretation of kinds, type families, and terms is defined in one big mutually recursive induction. We omit the details.

24.2 Proof-Theoretic Semantics

The proof-theoretical semantics uses additional judgments for equality of terms, type families, and kinds, and then uses an inference to axiomatize them.

The judgments are:

$\Gamma \vdash_\Sigma K \ = \ K'$	K and K' are equal kinds over Σ and Γ
$\Gamma \vdash_\Sigma A \ = \ A'$	A and A' are equal type families over Σ and Γ
$\Gamma \vdash_\Sigma t \ = \ t'$	t and t' are equal terms over Σ and Γ

The equality relation must be such that:

- Two kinds $A_1 \to \ldots \to A_m \to \texttt{type}$ and $A'_1 \to \ldots \to A'_n \to \texttt{type}$ can only be equal if $m = n$.
- Two type families can only be equal if their kinds are equal.

- Two terms can only be equal if their types are equal.

The inference system is given by:

- reflexivity, symmetry, transitivity for terms, type families, and kinds,
- congruence rules for all composed expressions:

$$\frac{\Gamma \vdash_\Sigma A \;=\; A' \quad \Gamma \vdash_\Sigma K \;=\; K'}{\Gamma \vdash_\Sigma A \to K \;=\; A' \to K'}\,eq\_kdfun$$

$$\frac{\Gamma \vdash_\Sigma A \;=\; A' \quad \Gamma \vdash_\Sigma t \;=\; t'}{\Gamma \vdash_\Sigma A\,t \;=\; A'\,t'}\,eq\_tpapp \qquad \frac{\Gamma \vdash_\Sigma A \;=\; A' \quad \Gamma, x : A \vdash_\Sigma B \;=\; B'}{\Gamma \vdash_\Sigma \Pi x : A.B \;=\; \Pi x : A'.B'}\,eq\_tpfun$$

$$\frac{\Gamma \vdash_\Sigma f \;=\; f' \quad \Gamma \vdash_\Sigma t \;=\; t'}{\Gamma \vdash_\Sigma f\,t \;=\; f'\,t'}\,eq\_termapp \qquad \frac{\Gamma \vdash_\Sigma A \;=\; A' \quad \Gamma, x : A \vdash_\Sigma t \;=\; t'}{\Gamma \vdash_\Sigma \lambda x : A.t \;=\; \lambda x : A'.t'}\,eq\_termlam$$

  The intuition of congruence is that composed expressions built from equal components must be equal. Note that every rule corresponds to a rule in Fig. 23.2.

- the rule

$$\frac{\Gamma \vdash_\Sigma E \;=\; E' \quad \Gamma \vdash_\Sigma F \;=\; F' \quad \Gamma \vdash_\Sigma E \;:\; F}{\Gamma \vdash_\Sigma E' \;:\; F'}$$

  where E, E' and F, F' are (i) terms and types, respectively, or (ii) type families and kinds, respectively. This rule says that typing respects equality. It can also be seen as a congruence rule for the typing judgment.

- β and η-equality for terms:

$$\frac{\Gamma \vdash_\Sigma (\lambda x : A.t)\; s \;:\; B}{\Gamma \vdash_\Sigma (\lambda x : A.t)\; s \;=\; t[x/s]}\,beta \qquad \frac{\Gamma \vdash_\Sigma f \;:\; \Pi x : A.B \quad x \text{ not in } \Gamma}{\Gamma \vdash_\Sigma f \;=\; \lambda x : A.(f\;x)}\,eta$$

Like many other type theories, the equality of LF enjoys the normalization property: For every expression (i.e., every kind, type family, or term), there is a normal form; every expression is equal to its normal form, and two expressions are equal iff their normal forms are identical (up to α-renaming of variables). Because the normal form can be computed, the equality judgments are decidable: To decide if two expressions are equal, we check if they have the same normal form.

24.3 The Relation between Model and Proof Theory

For type theories, we can define soundness and completeness as follows. Soundness: If two expressions are provably equal, then their interpretations are equal in every model and under every assignment. Completeness: If the interpretations of two expressions are equal in every model and under every assignment, then they are provably equal. In other words, soundness and completeness relate:

$$\Gamma \vdash_\Sigma E \;=\; E' \qquad \text{and} \qquad [\![\Gamma|E]\!]^{I,\alpha} = [\![\Gamma|E']\!]^{I,\alpha} \text{ for all } I, \alpha$$

for pairs (E, E') of kinds, type families, or terms.

Similarly to STT and HOL, we have soundness but not completeness for LF. To obtain completeness, we would need a larger collection of models, and the needed models are even weirder than the ones needed for STT and HOL.

Chapter 25

Representing Formal Languages in LF

LF is designed as a proof-theoretical logical framework. That means it is well-suited to represent the syntax and proof theory of logics. In that case, we call LF the meta-language and the encoded logic the object language. This distinction important to avoid confusion. The following is an overview how such an encoding works.

Object language	Meta-language LF
syntactic class	type (family) declaration
logical symbol	term declaration
expression	term
well-formedness of expression	typing judgment
judgment	type (family) declaration
inference rule	term declaration
derivation/proof	term
well-formedness of derivation	typing judgment

In particular, a whole logic L is represented as a signature Σ_L declaring syntactic classes, logical symbols, judgments, and inference rules. To actually have expressions to work with, we also need to represent signatures and contexts. This is done as follows:

Object language L	Meta-language LF
non-logical symbol	term declaration
signature Σ	signature Σ_L, Σ
signature morphism $\sigma : \Sigma \to \Sigma'$	signature morphism $id_{\Sigma_L}, \sigma \ : \ \Sigma_L, \Sigma \ \to \ \Sigma_L, \Sigma'$
axiom	term declaration (typed by a judgment)
theory (Σ, Θ)	signature Σ_L, Σ, Θ
theory morphism	signature morphism
variable	variable
assumption	variable
Σ-context and list of assumptions	Σ_L, Σ-context
substitution	substitution

25.1 Representing FOL

25.1.1 Syntax

For FOL, we have two syntactic classes: `term` and `form`. We represent them as two LF type declarations:

$$\texttt{term} : \texttt{type}, \qquad \texttt{form} : \texttt{type}$$

We have a bunch of logical symbols, e.g., $\wedge$ and $\forall$. We represent them as LF constant declarations:

$$\wedge : \texttt{form} \to \texttt{form} \to \texttt{form}, \qquad \forall : (\texttt{term} \to \texttt{form}) \to \texttt{form}$$

The former declares $\wedge$ to take two arguments, both formulas, and to return a formula. We will write $\wedge$ infix, i.e., $F \wedge G$ abbreviates $(\wedge\ F)\ G$. The latter declares $\forall$ to take one argument, a formula with a free term variable, and to return a formula.

The other connectives are declared in the same way where we use $\Rightarrow$ for implication:

$$\begin{aligned}
&\mathit{true} : \mathtt{form}\\
&\mathit{false} : \mathtt{form}\\
&\Rightarrow : \mathtt{form} \to \mathtt{form} \to \mathtt{form}\\
&\vee : \mathtt{form} \to \mathtt{form} \to \mathtt{form}\\
&\neg : \mathtt{form} \to \mathtt{form}\\
&\exists : (\mathtt{term} \to \mathtt{form}) \to \mathtt{form}
\end{aligned}$$

Let Σ_{FOL} be the LF signature declaring all these symbols.

The non-logical symbols are declared similarly. For function symbols f and predicate symbols p we put

$$f : \mathtt{term} \to \ldots \to \mathtt{term} \to \mathtt{term}, \qquad p : \mathtt{term} \to \ldots \to \mathtt{term} \to \mathtt{form}$$

where the dots indicate repetition according to the arity of the non-logical symbol. Let Σ be the thus-obtained LF encoding of some fixed FOL-signature.

FOL-variables are represented as LF-variables and FOL-contexts as LF-contexts. However, the LF-variables must be typed with their syntactic class. Thus, for all FOL-variables x in the context, the corresponding LF-context contains $x : \mathtt{term}$.

Now very FOL-expression – a FOL-term or a FOL-formula – can be written as an LF-term. The well-formedness of the expression is expressed by the LF typing judgments

$$\Gamma \vdash_{\Sigma_{\mathcal{FOL}},\,\Sigma} t \ : \ \mathtt{term}$$

and

$$\Gamma \vdash_{\Sigma_{\mathcal{FOL}},\,\Sigma} F \ : \ \mathtt{form}$$

These judgments look very similar to the well-formedness judgments that we have introduced before for FOL. This has two reasons: Firstly, the notation of FOL in these notes was chosen to match the encoding in LF. Secondly, the type theory of LF was chosen so that encodings are as seamless as possible.

25.1.2 Proof Theory

To encode the proof theory, we need one more judgment, namely for provability. We use the type family

$$\mathtt{proof} : \mathtt{form} \to \mathtt{type}$$

to holds proofs and thus to represent provability.

The intuition is that the LF typing judgment $p : \mathtt{proof}\ F$ represents the FOL-judgment "$\vdash\ F$ holds, and p is a derivation for it". In particular, derivations are represented as terms. This methodology is often summarized with *proofs-as-terms* or *judgments-as-types*. Therefore, we have to declare one LF-constant for every FOL-proof rule.

The rules for conjunction are represented like this:

$$\wedge I : \Pi F : \mathtt{form}.\Pi G : \mathtt{form}.(\mathtt{proof}\ F \to \mathtt{proof}\ G \to \mathtt{proof}\ (F \wedge G))$$

$$\wedge E_l : \Pi F : \mathtt{form}.\Pi G : \mathtt{form}.(\mathtt{proof}\ (F \wedge G) \to \mathtt{proof}\ F)$$

$$\wedge E_r : \Pi F : \mathtt{form}.\Pi G : \mathtt{form}.(\mathtt{proof}\ (F \wedge G) \to \mathtt{proof}\ G)$$

Intuitively, $\wedge I$ reads like this: "For all formulas F, for all formulas G, if p is a proof of F, and if q is a proof of G, then $\wedge I\ F\ G\ p\ q$ is a proof of $F \wedge G$. Alternatively, we can ignore the proofs themselves and only care about whether there is a proof of a formula. Then $\wedge I$ reads "For all formulas F, for all formulas G, if F is provable, and if G is provable, then $F \wedge G$ is provable. This corresponds exactly to the intended meaning of $\wedge I$.

The rules for implication look like this

$$\Rightarrow I : \Pi F : \texttt{form}.\Pi G : \texttt{form}.((\texttt{proof}\ F \to \texttt{proof}\ G) \to \texttt{proof}\ (F \Rightarrow G))$$

$$\wedge E : \Pi F : \texttt{form}.\Pi G : \texttt{form}.(\texttt{proof}\ (F \Rightarrow G) \to \texttt{proof}\ F \to \texttt{proof}\ G)$$

$\wedge E$ is straightforward. $\Rightarrow I$ uses a hypothetical judgment, i.e., an assumption that itself uses a local assumption; it reads “For all formulas F, for all formulas G, if f is a function transforming proofs of F into proofs of G, then $\Rightarrow I\ F\ G\ f$ is a proof of $F \Rightarrow G$”. Alternatively, without proofs, it reads: “For all formulas F, for all formulas G, if (if F is provable, then G is provable), then $F \Rightarrow G$ is provable.

The rules for universal quantification look like this

$$\forall I : \Pi F : \texttt{term} \to \texttt{form}.((\Pi x : \texttt{term}.\texttt{proof}\ (F\ x)) \to \texttt{proof}\ (\forall \lambda x : \texttt{term}.(F\ x)))$$

$$\forall E : \Pi F : \texttt{term} \to \texttt{form}.(\texttt{proof}\ (\forall \lambda x : \texttt{term}.(F\ x)) \to \Pi x : \texttt{term}.\texttt{proof}\ (F\ x))$$

$\forall I$ uses a parametric judgment, i.e., an assumption that depends on a free parameter (in this case x). It reads “For all formulas F with a free `term`-variable, if (for all `terms` x, $F(x)$ is provable), then $\forall x.F(x)$ is provable”. Note how the formula F with a free variable is represented as a term of type `term` $\to$ `form`; the formula $F(x)$ is represented as $(F\ x)$. $\forall E$ reads “For all formulas F with a free `term`-variable, if $\forall x.F(x)$ is provable, then for all `terms` x, $F(x)$ is provable.

A complete version of the encoding is given in Sect. 26.

The essence of the judgments-as-types representation can be summarized as in the table below: The object level is comprised of the things we talk about: terms, types, kinds, formulas – whatever syntactic classes the object language has. The judgment level contains how we talk about it: axioms, assumptions, proofs. In LF, the same formalism is used to represent both. This leads to an elegant unification of, e.g., signatures and theories.

Object language L		Meta-language LF
object level	judgment level	
syntactic class	judgment	type (family) declaration
non-logical symbol	axiom	constant declaration
signature	theory	signature
variable	assumption	variable declaration
context	list of assumptions	context
term/type/formula etc.	proof	term
well-formedness	provability	typing judgment

25.2 A General Recipe for the Representation of Inference Rules

In general, inference rules can be transformed into LF declarations using the following recipe.

1. Drop the arbitrary context Γ and the arbitrary list of assumption Δ – both are represented by LF contexts.
2. The rule is represented as an LF-constant whose name is the name of the rule, and whose type is determined as given below.
3. Take all free parameters of the rule and bind them in a Π-binder.
   (a) If the parameter may only use the variables from Γ, the type of the bound variable is the syntactic class of the parameter (which implies well-formedness of the parameter).
   (b) If the free parameter may use an additional variable (i.e., one that is not in Γ), its type is a function type where the domain is the syntactic class of the variable.

   Drop all side conditions regarding variable occurrences.
4. The scope of the thus-obtained Π-binder is a function type $J_1 \to \ldots \to J_n \to J$ where
   - $J_1, \ldots, J_n$ represent the assumptions of the rule
   - J represents the conclusion of the rule.

5. Each judgment is represented as follows

   (a) If the judgment only uses the variables from Γ and the assumptions from Δ, a base type, typically $\texttt{proof}\ F$ for some F.

   (b) If the judgment uses additional variables or assumptions, a function type $\Pi x_1 : C_1. \ldots \Pi C_n : E_n.\texttt{proof}\ F_1 \to \ldots \to \texttt{proof}\ F_n \to J$ where $x_1 : C_1, \ldots, x_n : C_n$ are the additional variables with their syntactic classes, $F_1, \ldots, F_n$ are the additional assumptions and J is the main part of the judgment.

We apply this recipe systematically to represent the rule

$$\frac{\Gamma;\Delta \vdash_\Sigma \exists x\ F \qquad \Gamma, x;\Delta, F \vdash_\Sigma G \qquad x \notin \Gamma}{\Gamma;\Delta \vdash_\Sigma G}\exists E$$

1. $\dfrac{\vdash_\Sigma \exists x\ F \qquad x; F \vdash_\Sigma G \qquad x \notin \Gamma}{\vdash_\Sigma G}\exists E$

2. $\exists E : \ldots$

3. The free parameters are F and G. We have the following conditions on variable occurrences: x may not occur in G, so the type of G is $\texttt{form}$. x may occur in F, so the type of F is $\texttt{term} \to \texttt{form}$. That yields

$$\exists E : \Pi F : \texttt{term} \to \texttt{form}.\Pi G : \texttt{form}. \ldots$$

4. We have two hypotheses J_1 and J_2 and one conclusion J. That yields:

$$\exists E : \Pi F : \texttt{term} \to \texttt{form}.\Pi G : \texttt{form}.J_1 \to J_2 \to J$$

5. We treat each judgment separately:

   - J_1 represents $\vdash_\Sigma \exists x F$. That yields $\texttt{proof}\ (\exists\ \lambda x : \texttt{term}.(F\ x))$. Note how $(F\ x)$ represents $F(x)$. Using the η-equality, we could also write $\texttt{proof}\ (\exists F)$.
   - J_2 represents $x; F \vdash_\Sigma G$ which uses one additional $\texttt{term}$-variable x and one additional assumption $F(x)$. That yields $\Pi x : \texttt{term}.\texttt{proof}\ (F\ x) \to \texttt{proof}\ G$.
   - J represents $\vdash_\Sigma G$, which yields $\texttt{proof}\ C$.

Finally, we obtain

$$\exists E : \Pi F : \texttt{term} \to \texttt{form}.\Pi G : \texttt{form}.\texttt{proof}\ (\exists\ \lambda x : \texttt{term}.(F\ x)) \to (\Pi x : \texttt{term}.\texttt{proof}\ (F\ x) \to \texttt{proof}\ G) \to \texttt{proof}\ C$$

25.3 Inheriting Rules from LF

One of the powerful features of LF is that the whole maintenance of the context – i.e., variables and assumptions – is provided by LF. Therefore, logic encodings in LF do not have to worry about them at all. In particular, all rules dealing with contexts and assumptions are automatically present and do not have to be represented as constants. These are:

- The rule saying that variables are well-formed terms. This rule is implied by the LF *termvar*.
- The axiom rule: We always have the term $\lambda x : \texttt{proof}\ F.x$. So we do not need a rule $axiom : \Pi F : \texttt{form}.\texttt{proof}\ F \to \texttt{proof}\ F$.
- The weakening rule: We always that $\Gamma \vdash_\Sigma t : A$ implies $\Gamma, x : A \vdash_\Sigma t : A$. Therefore, weakening rules such as

$$\frac{\Gamma;\Delta \vdash_\Sigma G}{\Gamma, x;\Delta \vdash_\Sigma G}$$

  or

$$\frac{\Gamma;\Delta \vdash_\Sigma G}{\Gamma;\Delta, F \vdash_\Sigma G}$$

  are always present in LF encodings.

- The cut rule: In LF, we can prove the substitution property: If $\Gamma \vdash_\Sigma s : A$ and $\Gamma, x : A \vdash_\Sigma t : B$, then also $\Gamma \vdash_\Sigma t[x/s] : B[x/s]$. Therefore, we always have the cut rule: Just imagine s to be the proof of the lemma A that is used in the proof t of B.

A small disadvantage is that LF is only appropriate for logics where these rules are actually sound. But this is the case for almost all important logics.

25.4 Representing STT and HOL

25.4.1 STT

STT has two syntactic classes: terms and types. The syntactic class of terms is subdivided into a separate class for each type. Therefore, we represent them in LF as

$$\mathtt{tp} : \mathtt{type}, \qquad \mathtt{tm} : \mathtt{tp} \to \mathtt{type}$$

Thus, we have the following correspondence of judgments

Object language STT	Meta-language LF
well-formed type A	term A of type $\mathtt{tp}$
well-formed term t of type A	term t of type $\mathtt{tm}\ A$
well-formed term t of type B with a free variable of type A	term t of type $\mathtt{tm}\ A \to \mathtt{tm}\ B$

Then we can add term declaration for the STT-expressions. Just like for FOL, there is one LF-term declaration for (almost) every inference rule for well-formed expressions:

- *tpbase*: Base types are declared explicitly as $a : \mathtt{tp}$
- *tpfun*: The binary type constructor for function types is declared as

$$\longrightarrow : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp}$$

  where we write $\longrightarrow$ for the arrow of STT to not confuse it with the $\to$ of LF. We will write $\longrightarrow$ infix.

- *termcon*: Constants of type A are declared as $c : \mathtt{tm}\ A$.
- *termvar*: The rule for variables is inherited from LF.
- *termapp*: We declare the binary application operator as

$$@ : \mathtt{tm}\ (A \longrightarrow B) \to \mathtt{tm}\ A \to \mathtt{tm}\ B$$

  We will write @ infix.

- *termlam*: We declare the λ-binder, which takes a term of type B with a free variable of type A, as

$$\mathtt{lam} : (\mathtt{tm}\ A \to \mathtt{tm}\ B) \to \mathtt{tm}\ (A \longrightarrow B)$$

25.4.2 Syntax of HOL

To represent HOL, we have to add some LF-declarations to the LF-declarations that represent STT. Namely:

$$o : \mathtt{tp}, \qquad \doteq : \Pi A : \mathtt{tp}.\mathtt{tm}\ A \to \mathtt{tm}\ A \to\ \mathtt{tm}\ o$$

The polymorphism of $\doteq$ that presented difficulties when representing HOL in STT is easily handled in LF: $\doteq$ takes three arguments, one for the type and then two for the terms of that type.

The representation of HOL gets more useful if we also represent all the abbreviations. This is not possible in core LF, but the Twelf implementation permits to give definitions to constants as well, see Sect. 26.

25.4.3 Proof Theory of HOL

For the proof theory, we start with the syntactic class of proofs or the provability judgment:

$$\mathtt{proof} : \mathtt{tm}\ o \to \mathtt{type}$$

Now we add the proof rules following the recipe from Sect. 25.2. As an example, we give the rule *eqlam*. The full encoding is given in Sect. 26.
For the rule

$$\frac{\Gamma,\ x : A; \Delta \vdash_\Sigma t \doteq_B t'}{\Gamma; \Delta \vdash_\Sigma (\lambda x : A.t) \doteq_{A\to B} (\lambda x : A.t')}\ eqlam$$

we proceed as follows:

1. Dropping Γ and Δ:
$$\frac{x : A \vdash_\Sigma t \doteq_B t'}{\vdash_\Sigma (\lambda x : A.t) \doteq_{A\to B} (\lambda x : A.t')}\ eqlam$$
2. The rule is a declaration $eqlam : \ldots$.
3. The free parameters are
   - A, which is an STT-type.
   - B, which is an STT-type.
   - t, which is an STT-term of STT-type B with a free variable of STT-type A.
   - t', which is an STT-term of STT-type B with a free variable of STT-type A.

   That yields $eqlam : \Pi A : \mathtt{tp}.\Pi B : \mathtt{tp}.\Pi t : \mathtt{tm}\ A \to \mathtt{tm}\ B.\Pi t : \mathtt{tm}\ A \to \mathtt{tm}\ B.\ldots$.
4. The rule has one assumption J_1 and a conclusion J.
5. We treat the judgments separately:
   - J_1 represents $x : A \vdash_\Sigma t \doteq_B t'$. To stress the occurrence of the variable x in t and t', we might write $x : A \vdash_\Sigma t(x) \doteq_B t(x)'$. Since the judgment uses an additional variable x of STT-type A, it is represented as $\Pi x : \mathtt{tm}\ A.\mathtt{proof}((t\ x) \doteq (t'\ x))$.
   - J represents $\vdash_\Sigma (\lambda x : A.t) \doteq_{A\to B} (\lambda x : A.t')$. This becomes
   $$\mathtt{proof}(\mathtt{lam}(\lambda x : \mathtt{tm}\ A.t\ x) \doteq \mathtt{lam}(\lambda x : \mathtt{tm}\ A.t'\ x))$$
   Note how every occurrence of the STT-λ is represented as $\mathtt{lam}$, and the binding of the variable x in an STT-expression as the LF-λ.

Putting everything together, we obtain

$$eqlam : \Pi A : \mathtt{tp}.\Pi B : \mathtt{tp}.\Pi t : \mathtt{tm}\ A \to \mathtt{tm}\ B.\Pi t : \mathtt{tm}\ A \to \mathtt{tm}\ B.$$
$$\big(\Pi x : \mathtt{tm}\ A.\mathtt{proof}((t\ x) \doteq (t'\ x))\big) \to \mathtt{proof}(\mathtt{lam}(\lambda x : \mathtt{tm}\ A.t\ x) \doteq \mathtt{lam}(\lambda x : \mathtt{tm}\ A.t'\ x))$$

In Twelf, after defining all the other connectives and quantifiers as abbreviations, we can continue and define all their proof rules as abbreviations.

25.5 Representing Foundations

See the paper [?].

Chapter 26

The LF Implementation Twelf

26.1 The Twelf Syntax for LF

Twelf is an implementation of LF ([?]).

LF	Twelf
`type`	`type`
$\Pi x : A.B$	`{x:A}B`
$A \to B$	`A -> B`
$\lambda x : A.B$	`[x:A]B`
$a : K$	`a:K.`
$c : A$	`c:A.`

Figure 26.1: Mathematical and ASCII Notation

Basic Declarations and Definitions The ASCII-based syntax of Twelf for declarations is given in Fig. 26.1. Our listings of Twelf code will use some syntax highlighting and display `->` as $\to$.

Twelf declarations may also give a definitions for a symbol as in `c:A=t.` if t is a term of type A. This means that c is an abbreviation for t

Furthermore, we group declarations into LF-signatures using `%sig S = {...}.`. Signatures may include other signatures using `%sig T = {%include S.}.`

Fixities and Omitting Brackets A symbol c taking two arguments may be declared infix using `%infix left 10 c.` where `left` and `10` handle the restoration of omitted brackets. The former is the associativity, i.e., $t_1\ c\ t_2\ c\ t_3$ is parsed as $(t_1\ c\ t_2)\ c\ t_3$. The alternatives are `right` and `none`. The latter is the precedence – constants with higher precedence bind stronger and brackets of stronger-binding constants can be omitted. For example, if we also have `%infix left 5 d.`, then $t_1\ c\ t_2\ d\ t_3$ is parsed as $(t_1\ c\ t_2)\ d\ t_3$. It is also possible to declare a constant that takes one argument to be prefix. For example, if we also have `%prefix 0 e`, then $e\ t_1\ c\ t_2$ is parsed as $e\ (t_1\ c\ t_2)$.

Type Reconstruction and Implicit Arguments Type reconstruction means that Twelf can find out the types of variables when the types are not given. Thus, users can often omit types of bound variables (which is sometimes good and sometimes for bad for readability).

Furthermore, we can omit outermost Π-binders altogether. For example, we can write $\doteq$: $\mathtt{tm}\ A \to \mathtt{tm}\ A \to \mathtt{tm}\ o$ rather than $\doteq$: $\Pi A : \mathtt{tp}.\mathtt{tm}\ A \to \mathtt{tm}\ A \to \mathtt{tm}\ o$. The omitted argument A is called an implicit argument. To make a free variable an implicit argument, the variable name must start with an upper case letter. When using the constant $\doteq$, only the explicit arguments are given – the value of the implicit arguments is computed automatically by Twelf. Together with an infix declaration, this has the effect that the equality of STT can be written intuitively as binary infix.

Comments Line comments start with `%%` or `% ` (percent-space). Range comments are of the form `%{ }%`. Commenting out everything that follows is done with `%.`.

Signatures and Signature Morphisms Twelf can also handle named signatures and signature morphisms ([?]). For example, to define two signatures A and B and a signature morphism c between them, we write

```
%sig A = {
  a : type.
  c : a.
}.
%sig B = {
  b : type.
  d : b.
}.
%view c : A -> B = {
  a := b -> b.
  c := [x:b]d.
}.
```

Signatures may include other signatures using `%include B.`, after which all symbols `s` of `B` are availalbe as `B..s`. If we use `%include B %open.`, the qualification can be omitted.

26.2 Representing FOL in Twelf

The full encoding of FOL is given as follows, where we use some syntax highlighting to enhance readability. Note how precedences are used to make negation bind stronger than the other connectives and to make `proof` bind weakest. That makes many brackets redundant.

```
%sig FOL = {
   term : type.
   form : type.

   proof  : form -> type.

   true  : form.
   false : form.
   and   : form -> form -> form.
   or    : form -> form -> form.
   imp   : form -> form -> form.
   not   : form -> form.
   eq    : term -> term -> form.

   forall : (term -> form) -> form.
   exists : (term -> form) -> form.

   %prefix 15 not.
   %infix left 10 and.
   %infix left 10 or.
   %infix none 10 imp.
   %infix none 20 eq.
   %prefix 10 forall.
   %prefix 10 exists.
   %prefix 0 proof.

   truthI : proof true.
   falseE : proof false -> proof F.
```

```
    notI : (proof F -> {G:form} proof G) -> proof not F.
   %% alternatively: (proof F -> proof false) -> proof not F.
    notE : proof not F -> proof F -> {F: form} proof F.
   %% alternatively: notE : proof not F -> proof F -> proof false.

    andI  : proof F -> proof G -> proof F and G.
    andEl : proof (F and G) -> proof F.
    andEr : proof (F and G) -> proof G.

    orIl : proof F -> proof F or G.
    orIr : proof G -> proof F or G.
    orE  : proof F or G -> (proof F -> proof H) -> (proof G -> proof H) -> proof H.

    impI : (proof F -> proof G) -> proof F imp G.
    impE : proof F imp G -> proof F -> proof G.

    forallI : {x:term} proof F x -> proof forall ([x:term] F x).
    forallE : proof forall ([x:term] F x) -> {x:term} proof F x.

    existsI : {t:term} proof F t -> proof exists ([x:term] F x).
    existsE : proof (exists [x:term] F x) -> ({t:term} proof F t -> proof H) -> proof H.

    nonempty : proof exists [x:term] true.
    tnd      : proof F or not F.
}.
```

26.3 Representing STT and HOL in Twelf

The full encoding of STT and HOL is given as follows. Note how abbreviations are used to define the connectives and quantifiers.

```
%sig STT = {

    tp : type.
   --> : tp -> tp -> tp.
   %infix right 10 -->.

   tm : tp -> type.
   %prefix 0 tm.
   @ : tm (A --> B) -> tm A -> tm B.
   %infix left 30 @.
   lam: (tm A -> tm B) -> tm A --> B.
}.

%sig HOL = {
  %include STT %open.
  o     : tp.
  =='   : tm A --> A --> o.
  ==    : tm A -> tm A -> tm o = [x: tm A] [y: tm A] ==' @ x @ y.
  %infix none 20 ==.

   true   : tm o = (lam [x: tm o] x) == (lam [x: tm o] x).
   forall': tm (A --> o) --> o = lam [f: tm A --> o] (f == (lam [x: tm A] true)).
   forall : (tm A -> tm o) -> tm o = [f : tm A -> tm o] forall' @ (lam [x: tm A] f x).
  %% now we can write forall ([x: tm A] f x) instead of forall' @ (lam [x: tm A] f x)
   false  : tm o = forall' @ (lam [x: tm o] x == true).
  ~'       : tm o --> o = lam ([x: tm o] x == false).
  ~        : tm o -> tm o = [x: tm o] ~' @ x.
  /\'      : tm o --> o --> o.  %%TODO
  /\       : tm o -> tm o -> tm o = [x: tm o] [y: tm o] /\' @ x @ y.  %infix left 10 /\.
  \/'      : tm o --> o --> o.  %%TODO
  \/       : tm o -> tm o -> tm o = [x][y] \/' @ x @ y.               %infix left 10 \/.
  =>'      : tm o --> o --> o.   %%TODO
  =>       : tm o -> tm o -> tm o = [x][y] =>' @ x @ y.               %infix none 10 =>.
  exists'  : tm (A --> o) --> o = lam [f: tm (A --> o)] (~ (forall [x : tm A] ~ (f @ x))).
  exists   : (tm A -> tm o) -> tm o = [f : tm A -> tm o] exists' @ (lam [x: tm A] f x).

  proof  : tm o -> type.
  %prefix 0 proof.

  eqrefl  : proof T == T.
  eqsym   : proof S == T -> proof T == S.
  eqapp   : proof F == G -> proof S == T -> proof (F @ S) == (G @ T).
  eqlam   : ({x: tm A} proof (T x) == (S x)) -> proof lam ([x: tm A] T x) == lam ([x: tm A] S

  beta    : proof (lam [x: tm A] T x) @ S == (T S).
  eta     : proof F == lam ([x: tm A] F @ x).

  eqI     : (proof F -> proof G) -> (proof G -> proof F) -> proof (F == G).
  eqE     : proof F == G -> proof F -> proof G.

  tnd     : proof forall [x: tm o] ((F == true) \/ (F == false)).
  %%alternatively: tndsplit : (proof F == true -> proof G) -> (proof F == false -> proof G) -

  nonempty : {A: tp} proof (exists' @ (lam [x: tm A] true)).

  form    : type = tm o.

  betao   : proof F X  -> proof (lam F) @ X = [p] (eqE (eqsym beta) p).

  trueI   : proof true. %%TODO
  falseE  : proof false -> proof F. %%TODO

  notI    : (proof F -> proof false) -> proof ~ F
          = [p: proof F -> proof false]
              (eqE (eqsym beta) (eqI p falseE)).
  notE    : proof ~ F -> proof F -> proof false
          = [p : proof ~ F] [q: proof F]
              (eqE (eqE beta p) q).

  %% TODO: missing rules

  forallI : ({x: tm A} proof F x) -> proof forall ([x: tm A] F x). %%TODO
  forallE : proof forall ([x: tm A] F x) -> {x: tm A} proof F x.%%TODO

  existsI : {t: tm A} proof (F t) -> proof exists ([x : tm A] F x)
            = [t][p] betao (notI [q: proof forall [x: tm A] ~ (lam F @ x)] (notE (forallE q t)
  existsE : (proof exists [x: tm A] F x) -> ({x : tm A} proof (F x) -> proof G) -> proof G. %
}.
```

Note how, e.g., equality is declared in two variants. First we declare the "official" equality, it is a family of constants $=='$ of STT-type $A \to A \to o$ (where A is an implicit argument) and thus LF-type `tm` $(A \longrightarrow A \longrightarrow o)$. The STT-term $s \doteq t$ is represented as the LF-term $==' \; @s@t$. This is of course awkward; therefore, we define $==$ of LF-type `tm` $A \to$ `tm` $A \ \to$ `tm` o as an abbreviation. Together with an infix declaration for ==, we can represent said STT-term as the much nicer-looking LF-term $s == t$.

Also note how proofs such a the one given in Lem. 13.2 is represented as an term: The LF-term can be read off from the proof by looking at the rule names. Because Twelf type-checks the definition, we are guaranteed that the proof is correct.

Appendix A

Mathematical Preliminaries

A.1 Power Sets

A.1.1 Subsets and Characteristic Functions

A.1.2 Operations on Subsets

A.1.3 Closures

Definition A.1 (Closure System)**.** A **closure system** C on a set A is a set of subsets of A (called the **closed** sets) such that the intersection of any collection of closed sets is again closed.

For any $X \subseteq A$, the smallest closed superset of X (which can be constructed by taking the intersection of all closed supersets of X) is called the C-**closure** of X.

Here "any number" is meant to include the intersection of

- infinitely many sets
- no set, i.e., A (which is the intersection of 0 subsets of A) is always closed

Definition A.2 (Closure Operator)**.** A **closure operator** C on a set A is a function from subsets of A to subsets of A such that

- for all $X \subseteq A$: $X \subseteq C(X)$
- for all $X \subseteq A$: $C(C(X)) = C(X)$
- for all $X, Y \subseteq A$: if $X \subseteq Y$, then $C(X) \subseteq C(Y)$

The sets for which $C(X) = X$ are called C-**closed**.

Theorem A.3 (Closure)**.** *Closure systems and closure operators are the same in the following sense:*

- *Given a closure operator, the closed sets form a closure system.*
- *Given a closure system, the closure is a closure operator.*

Closure operators occur in multiple places of these notes including

Example A.4. Given a set U, the reflexive/symmetric/transitive/reflexive-transitive/equivalence relations on U each form a closure system on $U \times U$. The respective closure operators are

- reflexive closure: $r \mapsto r \cup \Delta_U$
- symmetric closure: $r \mapsto r \cup r^{-1}$
- transitive closure: $r \mapsto r \cup rr \cup rrr \cup \ldots = \bigcup_{i=1} r^i$
- reflexive-transitive closure: $r \mapsto \Delta_U \cup r \cup rr \cup rrr \cup \ldots = \bigcup_{i=0} r^i$
- equivalence closure: $r \mapsto \bigcup_{i=0} (r \cup r^{-1})^i$

A.2 Relations and Functions

A binary relation between sets A and B is a subset $\# \subseteq A \times B$. We usually write $(x, y) \in \#$ as $x\#y$.

A.2.1 Classification

Definition A.5 (Properties of Relations)**.** We say that # is . . . if the following holds:

- functional: for all $x \in A$, there is at most one $y \in B$ such that $x\#y$.
- total: for all $x \in A$, there is at least one $y \in B$ such that $x\#y$.
- injective: for all $y \in B$, there is at most one $x \in A$ such that $x\#y$.
- surjective: for all $y \in B$, there is at least one $x \in A$ such that $x\#y$.

Moreover, # is . . . if it is . . . :

- a partial function: functional
- a non-deterministic function: total
- a (deterministic, total) function: functional and total
- bijective: total, functional, injective, and bijective

If a # is functional, we write $\#(x)$ for the $y \in B$ such that $x\#y$ if such a (necessarily unique) y exists.

Non-deterministic and partial functions can be awkward to work with mathematically. They are usually avoided or paraphrased. But they are important when programming: a partial function is simply a programming language function that may throw and exception for some arguments (e.g., division by 0), and a non-deterministic function is one that may return a different result each time it is called (e.g., a function to generate random numbers).

Because relations are the subsets of some set (namely $A \times B$), they admit all operations of subsets:

- empty relation: for no $x \in A$, $y \in B$ we have $x\#y$
- universal relation: for all $x \in A$, $y \in B$ we have $x\#y$
- for two relations r and s:
  - union of r and s: $x\,(r \cup s)\,y$ iff $x\,r\,y$ or $x\,r\,y$
  - intersection of r and s: $x\,(r \cup s)\,y$ iff $x\,r\,y$ and $x\,r\,y$

Moreover, if $r \subseteq s$ (if $x\,r\,y$, then also $x\,s\,y$), we say that s is a refinement of r.

A.2.2 Operations on Relations

Definition A.6 (Identity)**.** We define the identity relation Δ_A between A and A by:

$$x\,\Delta_A\,y \qquad \text{iff} \qquad x = y$$

i.e., $\Delta_A = \{(x, x) : x \in A\}$.

Theorem A.7 (Identity)**.** *The identity is functional, total, injective, and surjective.*

Definition A.8 (Composition)**.** Given relations r between A and B and s between B and C, we define the relation rs between A and C by

$$x\,(rs)\,z \qquad \text{iff} \qquad x\,r\,y \text{ and } y\,s\,z \text{ for some } y \in B$$

Composition rs is also written as $r; s$ or $s \circ r$.

Theorem A.9 (Composition)**.** *If two relations are both functional/total/injective/surjective, then so is their composition.*

Theorem A.10 (Properties of Composition)**.** *Composition is associative whenever defined.*
Identity is a neutral element for composition whenever defined.

Definition A.11 (Powers)**.** Given a relation r between A and A and a natural number n, we define X^n inductively by

$$r^0 = \Delta_A$$
$$r^{n+1} = X^n r$$

We also define

$$r^{-n} = (r^{-1})^n$$

Definition A.12 (Dual Relation)**.** For every relation $\#$, the relation $\#^{-1}$ is defined by $x \,\#^{-1}\, y$ iff $y \,\#\, x$. $\#^{-1}$ is called the **dual** of $\#$.

Theorem A.13 (Dual Relation)**.** *If a relation is functional/total/injective/surjective, then its dual is injective/-surjective/functional/total, respectively.*
In particular, the dual of a bijective function is a bijective function, which we call its ***inverse****.*

Theorem A.14 (Properties of Dualization)**.** *For all relations r, we have $(r^{-1})^{-1} = r$.*
For the identity relation, we have $\Delta_A^{-1} = \Delta_A$.
For all composed relations, we have $(rs)^{-1} = s^{-1}r^{-1}$.
For all powers of relations, we have $(r^n)^{-1} = (r^{-1})^n = r^{-n}$.

A.2.3 Point-Free Formulations

A statement about relations between A and B is called *point-free* if no elements $x \in A$ or $y \in B$ are mentioned explicitly. Point-free formulations take some getting used to but are more compact and eventually easier to work with.

Theorem A.15 (Point-Free Formulations)**.** *For a relation r between A and B, the following are equivalent:*

functional	$r^{-1}r \subseteq \Delta_B$
total	$rr^{-1} \supseteq \Delta_A$
injective	$rr^{-1} \subseteq \Delta_A$
surjective	$r^{-1}r \supseteq \Delta_B$

For a relation r on A, the following[1] *are equivalent:*

reflexive	$\Delta_A \subseteq r$
symmetric	$r^{-1} = r$
transitive	$rr \subseteq r$
reflexive and transitive	$\Delta_A \subseteq r$ *and* $rr = r$

Moreover, we have the following equalities for a function f[2]:

$$\ker f = f\,f^{-1} \qquad \operatorname{im} f \times \operatorname{im} f = f^{-1}\,f$$

A.3 Binary Relations on a Set

In this section, we consider a binary relation $\#$ on a set A, i.e., a subset $\# \subseteq A \times A$.

[1]The concepts on the left are defined in Def. A.16.
[2]See Def. .

A.3.1 Classification

Definition A.16 (Properties of Binary Relations)**.** We say that $\#$ is ... if the following holds:

- reflexive: for all x, $x\#x$
- irreflexive[3]: for no x, $x\#x$
- symmetric: for all x, y, if $x\#y$, then $y\#x$
- anti-symmetric[4]: for all x, y, if $x\#y$ and $y\#x$, then $x = y$ (= if $x\#y$ then not $y\#x$ (unless $x = y$))
- transitive: for all x, y, z, if $x\#y$ and $y\#z$, then $x\#z$

Moreover, we call $\#$ a ... if it is:

- strict order: irreflexive and transitive
- preorder: reflexive and transitive
- order[5]: preorder and anti-symmetric (= reflexive, transitive, and anti-symmetric)
- total[6]order: order and for all x, y, $x\#y$ or $y\#x$
- partial equivalence: symmetric and transitive (= not necessarily reflexive equivalence)
- equivalence: preorder and symmetric (= reflexive, transitive, and symmetric)

An element $a \in A$ is called ... of $\#$ if the following holds:

- least element: for all x, $a\#x$
- greatest element: for all x, $x\#a$
- least upper bound of x, y: $x\#a$ and $y\#a$ and for all z, if $x\#z$ and $y\#z$, then $a\#z$
- greatest lower bound of x, y: $a\#x$ and $a\#y$ and for all z, if $z\#x$ and $z\#y$, then $z\#a$

Theorem A.17 (Dual Relation)**.** *If a relation is reflexive/irreflexive/symmetric/antisymmetric/transitive/total, then so is its dual.*

If a is a least/greatest element for a relation, then it is a greatest/least element for its dual. If a is a least upper/greatest lower bound of x, y for some relation, then it is a greatest lower/least upper bound of x, y for the dual.

A.3.2 Equivalence Relations

Symbols Equivalence relations are usually written using infix symbols whose shape is reminiscent of horizontal lines, such as $=$, $\sim$, or $\equiv$. Often vertically symmetric symbols are used to emphasize the symmetry property.

Equivalence Classes and Quotients Equivalence relations allow grouping related elements into classes and collecting all the classes in what is called a quotient.

Definition A.18 (Quotient)**.** Consider a relation $\equiv$ on A. Then

- For $x \in A$, the set $\{y \in A \mid x \equiv y\}$ is called the (equivalence) **class** of x. It is often written as $[x]_\equiv$.
- $A/\equiv$ is the set of all classes. It is called the **quotient** of A by $\equiv$.

Definition A.19 (Kernel)**.** Consider a function $f : A \to B$. The **kernel** of f, written $\ker f$, is the binary relation on A defined by $x\,(\ker f)\,y$ iff $f(x) = f(y)$.

Definition A.20 (Partition)**.** A **partition** P on a set A is a set of non-empty, pairwise disjoint subsets of A whose overall union is A.

Theorem A.21. *For a relation $\equiv$ on A, the following are equivalent*[7]*:*

- *$\equiv$ is an equivalence.*
- *$\equiv$ is the kernel of some function, i.e., there is a set B and a function $f : A \to B$ such that $x \equiv y$ iff $f(x) = f(y)$.*
- *$A/\equiv$ is a partition on A, i.e., every element of A is in exactly one class in $A/\equiv$.*

[3]That's not the same as not being reflexive.
[4]That's not the same as not being symmetric.
[5]Orders are also called *partial order*, *poset* (for partially ordered set), or *ordering*.
[6]This notion of *total* has nothing to do with the one from Def. A.5 of the same name.

Partial Equivalence Relations Consider a partial equivalence relation $\equiv$ on A. $\equiv$ is not an equivalence because it is not reflexive. However, we can easily prove: if $x \equiv y$, then $x \equiv x$ and $y \equiv y$. Thus, the only elements for which $x \equiv x$ does not hold are the ones that are in relation to no element at all.

Thus, we have:

Theorem A.22. *A partial equivalence relation $\equiv$ on A is the same as an equivalence relation on a subset of A.*

Normal and Canonical Forms Instead of working with equivalence classes, we usually prefer working with representatives, i.e., designated elements of the classes that we use instead of the entire class.

Definition A.23 (System of Representatives)**.** Consider an equivalence relation $\equiv$ on A. A subset R of A is a system of **representatives** for $\equiv$ if it contains exactly one element from every $\equiv$-class.

Normal forms are used to choose representatives for each element:

Definition A.24 (Normal Forms)**.** Consider an equivalence relation $\equiv$ on A. A function $N : A \to A$ is called a …if

- **normal form**: $N(x) \equiv x$ and $N(N(x)) = N(x)$ for all $x \in A$
- **canonical form**: N is a normal form and $N(x) = N(y)$ whenever $x \equiv y$

We also call $N(x)$ the normal/canonical form of x. The process of mapping x to $N(x)$ is called **normalization**.

The main application of canonical forms is that we can check $x \equiv y$ by comparing $N(x)$ and $N(y)$.

Theorem A.25. *Consider an equivalence relation $\equiv$ on A and a normal form $N : A \to A$. The following are equivalent:*

- *N is a canonical form.*
- *The image of N is a system of representatives.*
- *$\equiv$ is the kernel of N.*

A.3.3 Orders

Theorem A.26 (Strict Order vs. Order)**.** *For every strict order $<$ on A, the relation "$x < y$ or $x = y$" is an order.*

For every order $\leq$ on A, the relation "$x \leq y$ and $x \neq y$" is a strict order.

Thus, strict orders and orders come in pairs that carry the same information.

Strict orders are usually written using infix symbols whose shape is reminiscent of a semi-circle that is open to the right, such as $<$, $\subset$, or $\prec$. This emphasizes the anti-symmetry ($x < y$ is very different from $y < x$.) and the transitivity ($< \ldots <$ is still $<$.) The corresponding order is written with an additional horizontal bar at the bottom, i.e., $\leq$, $\subseteq$, or $\preceq$. In both cases, the mirrored symbol is used for the dual relation, i.e., $>$, $\supset$, or $\succ$, and $\geq$, $\supseteq$, and $\succeq$.

Theorem A.27. *If $\leq$ is an order, then least element, greatest element, least upper bound of x, y, and greatest lower bound of x, y are unique whenever they exist.*

Theorem A.28 (Preorder vs. Order)**.** *For every preorder $\leq$ on A, the relation "$x \leq y$ and $y \leq x$" is an equivalence.*

For equivalence classes X and Y of the resulting quotient, $x \leq y$ holds for either all pairs or no pairs $(x, y) \in X \times Y$. If it holds for all pairs, we write $X \leq Y$.

The relation $\leq$ on the quotient is an order.

[7]Logical equivalence is itself an equivalence relation.

Remark A.29 (Order vs. Total Order). If $\leq$ is a preorder, then for all elements x, y, there are four mutually exclusive options:

	$x \leq y$	$x \geq y$	$x = y$
x strictly smaller than y, i.e., $x < y$	true	false	false
x strictly greater than y, i.e., $x > y$	false	true	false
x and y incomparable	false	false	false
x and y similar	true	true	maybe

Now anti-symmetry excludes the option of similarity (except when $x = y$ in which case trivially $x \leq y$ and $x \geq y$). And totality excludes the option of incomparability.

Combining the two exclusions, a total order only allows for $x > y$, $y < x$, and $x = y$.

A.4 Binary Functions on a Set

A binary function on A is a function $\circ : A \times A \to A$. We usually write $\circ(x, y)$ as $x \circ y$.

Definition A.30 (Properties of Binary Functions)**.** We say that $\circ$ is … if the following holds:

- associative: for all x, y, z, $x \circ (y \circ z) = (x \circ y) \circ z$
- commutative: for all x, y, $x \circ y = y \circ x$
- idempotent: for all x, $x \circ x = x$

An element $a \in A$ is called a … element of $\circ$ if the following holds:

- left-neutral: for all x, $a \circ x = x$
- right-neutral: for all x, and $x \circ a = x$
- neutral: left-neutral and right-neutral
- left-absorbing: for all x, $a \circ x = a$
- right-absorbing: for all x, $x \circ a = a$
- absorbing: left-absorbing and right-absorbing
- if e is a neutral element:
  - left-inverse of x: $a \circ x = e$
  - right-inverse of x: $x \circ a = e$
  - inverse of x: left-inverse and right-inverse of x

Moreover, we say that $\circ$ is a … if it is/has:

- semigroup: associative
- monoid: associative and neutral element
- group: monoid and inverse elements for all x
- semilattice: associative, commutative, and idempotent
- bounded semilattice: semilattice and neutral element

Terminology A.31. The terminology for *absorbing* is not well-standardized. *Attractive* is an alternative word sometimes used instead.

Theorem A.32. *Neutral and absorbing element of $\circ$ are unique whenever they exist.*

If $\circ$ is a monoid, then the inverse of x is unique whenever it exists.

A.5 The Integer Numbers

A.5.1 Divisibility

Definition A.33 (Divisibility)**.** For $x, y \in \mathbb{Z}$, we write $x|y$ iff there is a $k \in \mathbb{Z}$ such that $x * k = y$.

We say that y is divisible by x or that x divides y.

Remark A.34 (Divisible by 0 and 1). Even though division by 0 is forbidden, the case $x = 0$ is perfectly fine. But it is boring: $0|x$ iff $x = 0$.
Similarly, the case $x = 1$ is trivial: $1|x$ for all x.

Theorem A.35 (Divisibility). *Divisibility has the following properties for all $x, y, z \in Z$*

- *reflexive: $x|x$*
- *transitive: if $x|y$ and $y|z$ then $x|z$*
- *anti-symmetric for natural numbers $x, y \in \mathbb{N}$: if $x|y$ and $y|x$, then $x = y$*
- 1 *is a least element:* $1|x$
- 0 *is a greatest element:* $x|0$
- $\gcd(x, y)$ *is a greatest lower bound of* x, y
- $\text{lcm}(x, y)$ *is a least upper bound of* x, y

Thus, $|$ is a preorder on $\mathbb{Z}$ and an order on $\mathbb{N}$.

Divisibility is preserved by arithmetic operations: If $x|m$ and $y|m$, then

- *preserved by addition: $x + y|m$*
- *preserved by subtraction: $x - y|m$*
- *preserved by multiplication: $x * y|m$*
- *preserved by division if $x/y \in \mathbb{Z}$: $x/y|m$*
- *preserved by negation of any argument: $-x|m$ and $x| - m$*

gcd *has the following properties for all $x, y \in \mathbb{N}$:*

- *associative:* $\gcd(\gcd(x, y), z) = \gcd(x, \gcd(y, z))$
- *commutative:* $\gcd(x, y) = \gcd(y, x)$
- *idempotent:* $\gcd(x, x) = x$
- 0 *is a neutral element:* $\gcd(0, x) = x$
- 1 *is an absorbing element:* $\gcd(1, x) = 1$

lcm *has the same properties as* gcd *except that* 1 *is neutral and* 0 *is absorbing.*

Theorem A.36. *For all $x, y \in \mathbb{Z}$, there are numbers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.*
a and b can be computed using the extended Euclidean algorithm.

Definition A.37. If $\gcd(x, y) = 1$, we call x and y **coprime**.
For $x \in \mathbb{N}$, the number of coprime $y \in \{0, \ldots, x - 1\}$ is called $\varphi(x)$. φ is called Euler's **totient function**.

Example A.38. We have $\varphi(0) = 0$, $\varphi(1) = \varphi(2) = 1$, $\varphi(3) = \varphi(4) = 2$, and so on. Because $\gcd(x, 0) = x$, we have $\varphi(x) \leq x - 1$. x is prime iff $\varphi(x) = x - 1$.

A.5.2 Equivalence Modulo

Definition A.39 (Equivalence Modulo). For $x, y, m \in \mathbb{Z}$, we write $x \equiv_m y$ iff $m|x - y$.

Theorem A.40 (Relationship between Divisibility and Modulo). *The following are equivalent:*

- $m|n$
- $\equiv_m \supseteq \equiv_n$ *(i.e., for all x, y we have that $x \equiv_n y$ implies $x \equiv_m y$)*
- $n \equiv_m 0$

Remark A.41 (Modulo 0 and 1). In particular, the cases $m = 0$ and $m = 1$ are trivial again:

- $x \equiv_0 y$ iff $x = y$,
- $x \equiv_1 y$ always

Thus, just like 0 and 1 are greatest and least element for $|$, we have that $\equiv_0$ and $\equiv_1$ are the smallest and the largest equivalence relation on $\mathbb{Z}$.

Theorem A.42 (Modulo)**.** *The relation $\equiv_m$ has the following properties*

- *reflexive: $x \equiv_m x$*
- *transitive: if $x \equiv_m y$ and $y \equiv_m z$ then $x \equiv_m z$*
- *symmetric: if $x \equiv_m y$ then $y \equiv_m x$*

Thus, it is an equivalence relation.

It is also preserved by arithmetic operations: If $x \equiv_m x'$ and $y \equiv_m y'$, then

- *preserved by addition: $x + y \equiv_m x' + y'$*
- *preserved by subtraction: $x - y \equiv_m x' - y'$*
- *preserved by multiplication: $x \cdot y \equiv_m x' \cdot y'$*
- *preserved by division if $x/y \in \mathbb{Z}$ and $x'/y' \in \mathbb{Z}$: $x/y \equiv_m x'/y'$*
- *preserved by negation of both arguments: $-x \equiv_m -x'$*

A.5.3 Arithmetic Modulo

Definition A.43 (Modulus)**.** We write $x \bmod m$ for the smallest $y \in \mathbb{N}$ such that $x \equiv_m y$.
We also write $modulus_m$ for the function $x \mapsto x \bmod m$. We write $\mathbb{Z}_m$ for the image of $modulus_m$.

Theorem A.44 (Modulus)**.** *$modulus_m$ and $\mathbb{Z}_m$ are a canonical form and a system of representatives for $\equiv_m$.*

Remark A.45 (Modulo 0 and 1). The cases $m = 0$ and $m = 1$ are trivial again:

- $x \bmod 0 = x$ and $\mathbb{Z}_0 = \mathbb{Z}$
- $x \bmod 1 = 0$ and $\mathbb{Z}_1 = \{0\}$

Remark A.46 (Possible Values). For $m \neq 0$, we have $\mathbb{Z}_m = \{0, \ldots, m-1\}$. In particular, there are m possible values for $x \bmod m$.
For example, we have $x \bmod 1 \in \{0\}$. And we have $x \bmod 2 = 0$ if x is even and $x \bmod 2 = 1$ if x is odd.

Definition A.47 (Arithmetic Modulo m)**.** For $x, y \in \mathbb{Z}$, we define arithmetic operations modulo m by

$$x \circ_m y = (x \circ y) \bmod m \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, if there is a unique $q \in \mathbb{Z}_m$ such that $q \cdot x \equiv_m y$, we define $x/_m y = q$.

Note that the condition $y|x$ is neither necessary nor sufficient for $x/_m y$ to de defined. For example, $2/_4 2$ is undefined because $1 \cdot 2 \equiv_4 3 \cdot 2 \equiv_4 2$. Conversely, $2/_4 3$ is defined, namely 2.

Theorem A.48 (Arithmetic Modulo m)**.** *For $x, y \in \mathbb{Z}$,* mod *commutes with arithmetic operations in the sense that*

$$(x \circ y) \bmod m = (x \bmod m) \circ_m (y \bmod m) \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, $x/_m y$ is defined iff $\gcd(y, m) = 1$ and

$$(x/y) \bmod m = (x \bmod m)/_m (y \bmod m) \qquad \text{if} \qquad y|x$$

$$x/_m y = x \cdot_m a \qquad \text{if } ay + bm = 1 \textit{ as in Thm. A.36}$$

Theorem A.49 (Fermat's Little Theorem)**.** *For all prime numbers p and $x \in \mathbb{Z}$, we have that $x^p \equiv_p x$.*
If x and p are coprime, that is equivalent to $x^{p-1} \equiv_p 1$.

A.5.4 Digit-Base Representations

Fix $m \in \mathbb{N} \setminus \{0\}$, which we call the base.

Theorem A.50 (Div-Mod Representation)**.** *Every $x \in \mathbb{Z}$ can be uniquely represented as $a \cdot m + b$ for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_m$.*
Moreover, $b = x \bmod m$. We write $b \operatorname{div} m$ for a.

Definition A.51 (Base-m-Notation)**.** For $d_i \in \mathbb{Z}_m$, we define $(d_k \ldots d_0)_m = d_k \cdot m^k + \ldots + d_1 \cdot k + d_0$.
The d_i are called **digits**.

Theorem A.52 (Base-m Representation)**.** *Every $x \in \mathbb{N}$ can be uniquely represented as $(0)_m$ or $(d_k \ldots d_0)_m$ such that $d_k \neq 0$.*
Moreover, we have $k = \lfloor \log_m x \rfloor$ and $d_0 = x \bmod m$, $d_1 = (x \operatorname{div} m) \bmod m$, $d_2 = ((x \operatorname{div} m) \operatorname{div} m) \bmod m$ and so on.

Example A.53 (Important Bases). We call $(d_k \ldots d_0)_m$ the binary/octal/decimal/hexadecimal representation if $m = 2, 8, 10, 16$, respectively.
In case $m = 16$, we write the elements of $\mathbb{Z}_m$ as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

A.5.5 Finite Fields

In this section, let $m = p$ be prime.

Construction Because p is prime, $x/_p y$ is defined for all $x, y \in \mathbb{Z}_p$ with $y \neq 0$. Moreover, $\mathbb{Z}_p$ is a field.
Up to isomorphism, all finite fields are obtained as n-dimensional vector spaces $\mathbb{Z}_p^n$ for some prime p and $n \geq 1$. This field is usually called F_{p^n} because it has p^n elements. From now on, let $q = p^n$.
The elements of F_q are vectors $(a_0, \ldots, a_{n-1})$ for $a_i \in \mathbb{Z}_p$. Addition and subtraction are component-wise, the 0-element is $(0, \ldots, 0)$, the 1-element is $(1, 0, \ldots, 0)$.
However, multiplication in F_q is tricky if $n > 1$. To multiply two elements, we think of the vectors $(a_0, \ldots, a_{n-1})$ as polynomials $a_{n-1}X^{n-1} + \ldots + a_1X + a_0$ and multiply the polynomials. This can introduce powers X^n and higher, which we eliminate using $X^n = k_{n-1}X^{n-1} + \ldots + k_1X + k_0$ for certain k_i. The resulting polynomial has degree at most $n - 1$, and its coefficients (modulo p) yield the result.
The values k_i always exists but are non-trivial to find. They must be such that the polynomial $X^n - k_{n-1}X^{n-1} - \ldots - k_1X - k_0$ has no roots in $\mathbb{Z}_p$. There may be multiple such polynomials, which may lead to different multiplication operations. However, all of them yield isomorphic fields.

Binary Fields The operations become particularly easy if $p = 2$. The elements of F_{2^n} are just the bit vectors of length n. Addition and subtraction are the same operation and can be computed by component-wise XOR. Multiplication is a bit more complex but can be obtained as a sequence of bit-shifts and XORs.

Exponentiation and Logarithm Because F_q has multiplication, we can define natural powers in the usual way:

Definition A.54. For $x \in F_q$ and $l \in \mathbb{N}$, we define $x^l \in F_q$ by $x^0 = 1$ and $x^{l+1} = x \cdot x^l$.
If $l \in \mathbb{N}$ is the smallest number such that $x^l = y$, we write $l = \log_x y$ and call n the **discrete q-logarithm** of y with base x.

The powers $1, x, x^2, \ldots \in F_q$ of x can take only $q - 1$ different values because F_q has only q elements and x^l can never be 0 (unless $x = 0$). Therefore, they must be periodic:

Theorem A.55. *For every $x \in F_q$, we have $x^q = x$. If $x \neq 0$, that is equivalent to $x^{q-1} = 1$.*

For some x, the period is indeed $q-1$, i.e., we have $\{1, x, x^2, \dots, x^{q-1}\} = F_q \setminus \{0\}$. Such an x is called a **primitive element** of F_q. In that case $\log_x y$ is defined for all y.

But the period may be smaller. For example, the powers of 1 are $1, \dots, 1$, i.e., 1 has period 1. For a non-trivial example consider $p = 5$, $n = 1$, (i.e., $q = 5$): The powers of 4 are $4^0 = 1$, $4^1 = 4$, $4^2 = 16 \bmod 5 = 1$, and $4^3 = 4$.

If the period is smaller than $q-1$, x^l does not take all possible values in F_q. In that case $\log_x y$ is not defined for all $y \in F_q$.

Computing x^l is straightforward and can be done efficiently. (If $n > 1$, we first have to find the values k_i needed to do the multiplication, but we can precompute them once and for all.)

Determining whether $\log_x y$ is defined and computing its value is also straightforward: We can enumerate all powers $1, x, x^2, \dots$ until $x^l = 1$ (in which case the logarithm is undefined) or $x^l = y$ (in which case the logarithm is l). However, no efficient algorithm is known.

A.5.6 Infinity

Occasionally, it is useful to compute also with infinity ∞ or $-\infty$. When adding infinity, some but not all arithmetic operations still behave nicely.

Positive Infinity We write $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

The order $\leq$ works as usual. ∞ is the greatest element.

Addition works as usual. ∞ is an attractive element.

Subtraction is introduced as usual, i.e., $a - b = x$ whenever x is the unique value such that $a = x + b$. Thus, $\infty - n = \infty$ for $n \in \mathbb{N}$. $x - \infty$ is undefined. The law $x - x = 0$ does not hold anymore.

Multiplication becomes partial because $\infty \cdot 0$ is undefined. For $x \neq 0$, we put $\infty \cdot x = \infty$.

Divisibility $|$ is defined as usual. Thus, we have $x | \infty$ for all $x \neq 0$, and $\infty | x$ iff $x = \infty$. There is no greatest element anymore because: 0 and ∞ are both greater than every other element except for each other.

Negative Infinity We write $\mathbb{Z}^\infty = \mathbb{Z} \cup \{\infty, -\infty\}$.

The order $\leq$ works as usual. $-\infty$ is the least and ∞ the greatest element.

Addition becomes partial because $-\infty + \infty$ is undefined. We put $-\infty + z = -\infty$ for $z \neq \infty$.

Subtraction is introduced as usual. Thus, $z - \infty = -\infty - z = -\infty$ for $z \in \mathbb{Z}$. $\infty - \infty$ is undefined.

Multiplication works similarly to $\mathbb{N}^\infty$. $-\infty \cdot 0$ is undefined. And for $x \neq 0$, we define $\infty \cdot x$ and $-\infty \cdot x$ as ∞ or $-\infty$ depending on the signs.

A.6 Size of Sets

The size $|S|$ of a set S is a very complex operation because there are different degrees of infinity, i.e., not all infinite sets have the same size. Specifically, we have that $|\mathcal{P}(S)| > |S|$, i.e., we have infinitely many degrees of infinity.

In computer science, we are only interested in countable sets. Therefore, we use a much simpler definition of size: we write C for *countable* and U for uncountable, i.e., everything that is bigger:

Definition A.56 (Size of sets)**.** The size $|S| \in \mathbb{N} \cup \{C, U\}$ of a set S is defined by:

- if S is finite: $|S|$ is the number of elements of S
- if S is infinite and bijective to $\mathbb{N}$: $|S| = C$, and we say that S is **countable**
- if S is infinite and not bijective to $\mathbb{N}$: $|S| = U$, and we say that S is **uncountable**

We can compute with set sizes as follows:

Definition A.57 (Computing with Sizes)**.** For two sizes $s, t \in \mathbb{N} \cup \{C, U\}$, we define addition, multiplication, and exponentiation by the following tables:

	$s+t$	t: $n \in \mathbb{N}$	C	U
	$m \in \mathbb{N}$	$m+n$	C	U
s	C	C	C	U
	U	U	U	U

	$s*t$	t: $n \in \mathbb{N}$	C	U
	$m \in \mathbb{N}$	$m*n$	C	U
s	C	C	C	U
	U	U	U	U

	s^t	t: 0	1	$n \in \mathbb{N} \setminus \{0\}$	C	U
	0	1	0	0	0	0
	1	1	1	1	1	1
s	$m \in \mathbb{N} \setminus \{0\}$	1	m	m^n	U	U
	C	1	C	C	U	U
	U	1	U	U	U	U

Because exponentiation s^t is not commutative, the order matters: s is given by the row and t by the column.

The intuition behind these rules is given by the following:

Theorem A.58. *For all sets S, T, we have for the size of the*

- *disjoint union:*
$$|S \uplus T| = |S| + |T|$$
- *Cartesian product:*
$$|S \times T| = |S| * |T|$$
- *set of functions from T to S:*
$$|S^T| = |S|^{|T|}$$

Thus, we can understand the rules for exponentiation as follows. Let us first consider the 4 cases where one of the arguments has size 0 or 1: For every set A

1. there is exactly one function from the empty set (namely the empty function): $|A^{\varnothing}| = 1$,
2. there are as many functions from a singleton set as there are elements of A: $|A^{\{x\}}| = |A|$,
3. there are no functions to the empty set (unless A is empty): $|\varnothing^A| = 0$ if $A \neq \varnothing$,
4. there is exactly one function into a singleton set (namely the constant function): $|\{x\}^A| = 1$,

Now we need only one more rule: The set of functions from a non-empty finite set to a finite/countable/uncountable set is again finite/countable/uncountable. In all other cases, the set of functions is uncountable.

A.7 Important Sets and Functions

The meaning and purpose of a data structure is to describe a set in the sense of mathematics. Similarly, the meaning and purpose of an algorithm is to describe a function between two sets.

Thus, it is helpful to collect some sets and functions as examples. These are typically among the first data structures and algorithms implemented in any programming language and they serve as test cases for evaluating our languages.

A.7.1 Base Sets

When building sets, we have to start somewhere with some sets that are assumed to exist. These are called the *bases sets* or the *primitive sets*.

The following table gives an overview of commonly used base sets, where we also list the size of each set according to Def. A.56:

set	description/definition	size
	typical base sets of mathematics[8]	
$\varnothing$	empty set	0
$\mathbb{N}$	natural numbers	C
$\mathbb{Z}$	integers	C
$\mathbb{Z}_m$ for $m > 0$	integers modulo m, $\{0, \ldots, m-1\}$ [9]	m
$\mathbb{Q}$	rational numbers	C
$\mathbb{R}$	real numbers	U
	additional or alternative base sets used in computer science	
void	alternative name for $\varnothing$	0
unit	unit type, $\{()\}$, equivalent to $\mathbb{Z}_1$	1
$\mathbb{B}$	booleans, $\{false, true\}$, equivalent to $\mathbb{Z}_2$	2
int	primitive integers, $-2^{n-1}, \ldots, 2^{n-1}-1$ for machine-dependent n, equivalent to $\mathbb{Z}_{2^n}$[10]	2^n
float	IEEE floating point approximations of real numbers	C
char	characters	finite[11]
string	lists of characters	C

A.7.2 Functions on the Base Sets

For every base set, we can define some basic operations. These are usually built-in features of programming languages whenever the respective base set is built-in.

We only list a few examples here.

Numbers

For all number sets, we can define addition, subtraction, multiplication, and division in the usual way.

Some care must be taken when subtracting or dividing because the result may be in a different set. For example, the difference of two natural numbers is not in general a natural number but only an integer (e.g., $3 - 5 \notin \mathbb{N}$). Moreover, division by 0 is always forbidden.

Quotients of the Integers

The function $modulus_m$ (see Sect. A.5.3) for $m \in \mathbb{N}$ maps $x \in \mathbb{Z}$ to $x \bmod m \in \mathbb{Z}_m$.

In programming languages, the set $\mathbb{Z}_m$ is usually not provided. Instead, $x \bmod y$ is built-in as a function on *int*. [12]

Booleans

On booleans, we can define the usual boolean operations conjunction (usually written & or &&), disjunction (usually written | or ||), and negation (usually written !).

Moreover, we have the equality and inequality functions, which take two objects x, y and return a boolean. These are usually written $x == y$ and $x \mathrel{!=} y$ in text files and $x = y$ and $x \neq y$ on paper.

A.7.3 Set Constructors

From the base sets, we build all other sets by applying set constructors. Those are operations that take sets and return new sets.

[8]All of mathematics can be built by using $\varnothing$ as the only base set because the others are definable. But it is common to assume at least the number sets as primitives.

[9]$\mathbb{Z}_0$ also exists but is trivial: $\mathbb{Z}_0 = \mathbb{Z}$.

[10]Primitive integers are the 2^n possible values for a sequence of n bits. Old machines used $n = 8$ (and the integers were called "bytes"), later machines used $n = 16$ (called "words"). Modern machines typically use 32-bit or 64-bit integers. Modern programmers usually—but dangerously—assume that 2^n is much bigger than any number that comes up in practice so that essentially (but not actually) $int = \mathbb{Z}$. Some programming languages (e.g., Python) correctly implement $int = \mathbb{Z}$.

[11]The ASCII standard defined 2^7 or 2^8 characters. Nowadays, we use Unicode characters, which is a constantly growing set containing the characters of virtually any writing system, many scientific symbols, emojis, etc. Many programming languages assume that there is one character for every primitive integers, e.g., typically 2^{32} characters.

[12]Some care must be taken if x is negative because not all programming languages agree.

The following table gives an overview of commonly used set constructors, where we also list the size of each set according to Def. A.57:

set	description/definition	size
	typical constructors in mathematics	
$A \uplus B$	disjoint union	$\|A\| + \|B\|$
$A \times B$	(Cartesian) product	$\|A\| * \|B\|$
A^n for $n \in \mathbb{N}$	n-dimensional vectors over A	$\|A\|^n$
B^A or $A \to B$	functions from A to B	$\|B\|^{\|A\|}$
$\mathcal{P}(A)$	power set, equivalent to $\mathbb{B}^A$	$2^{\|A\|} = \begin{cases} 2^n & \text{if } \|A\| = n \\ U & \text{otherwise} \end{cases}$
$\{x \in A \| P(x)\}$	subset of A given by property P	$\leq \|A\|$
$\{f(x) : x \in A\}$	image of operation f when applied to elements of A	$\leq \|A\|$
A/r	quotient set for an equivalence relation r on A	$\leq \|A\|$
	selected additional constructors often used in computer science	
A^*	lists over A	$\begin{cases} 1 & \text{if } \|A\| = 0 \\ U & \text{if } \|A\| = U \\ C & \text{otherwise} \end{cases}$
$A^?$	optional element[13] of A	$1 + \|A\|$
	for new names $l_1, \ldots, l_n$	
$enum\{l_1, \ldots, l_n\}$	enumeration: like $\mathbb{Z}_n$ but also introduces named elements l_i of the enumeration	n
$l_1(A_1) \| \ldots \| l_n(A_n)$	labeled union: like $A_1 \uplus \ldots \uplus A_n$ but also introduces named injections l_i from A_i into the union	$\|A_1\| + \ldots + \|A_n\|$
$\{l_1 : A_1, \ldots, l_n : A_n\}$	record: like $A_1 \times \ldots \times A_n$ but also introduces named projections l_i from the record into A_i	$\|A_1\| * \ldots * \|A_n\|$
inductive data types[14]		C
classes[15]		U

A.7.4 Characteristic Functions of the Set Constructors

Every set constructor comes systematically with characteristic functions into and out of the constructed sets S. These functions allow building elements of S or using elements of S for other computations.

For some sets, these functions do not have standard notations in mathematics. In those cases, different programming languages may use slightly different notations.

The following table gives an overview:

set C	build an element of C	use an element x of C
$A_1 \uplus A_2$	$inj_1(a_1)$ or $inj_2(a_2)$ for $a_i \in A_i$	pattern-matching
$A_1 \times A_2$	(a_1, a_2) for $a_i \in A_i$	$x.i \in A_i$ for $i = 1, 2$
A^n	$(a_1, \ldots, a_n)$ for $a_i \in A$	$x.i \in A$ for $i = 1, \ldots, n$
B^A	$(a \in A) \mapsto b(a)$	$x(a)$ for $a \in A$
A^*	$[a_0, \ldots, a_{l-1}]$[16] for $a_i \in A$	pattern-matching
$A^?$	$None$ or $Some(a)$ for $a \in A$	pattern-matching
$enum\{l_1, \ldots, l_n\}$	l_1 or … or l_n	switch statement or pattern-matching
$l_1(A_1) \| \ldots \| l_n(A_n)$	$l_1(a_1)$ or … or $l_n(a_n)$ for $a_i \in A_i$	pattern-matching
$\{l_1 : A_1, \ldots, l_n : A_n\}$	$\{l_1 = a_1, \ldots, l_n = a_n\}$ for $a_i \in A_i$	$x.l_i \in A_i$
inductive data type A	$l(u_1, \ldots, u_n)$ for a constructor l of A	pattern-matching
class A	**new** A	$x.l(u_1, \ldots, u_n)$ for a field l of A

[13] An optional element of A is either absent or an element of A.

[14] These are too complex to define at this point. They are a key feature of functional programming languages like SML.

[15] These are too complex to define at this point. They are a key feature of object-oriented programming languages like Java.

[16] Mathematicians start counting at 1 and would usually write a list of length n as $[a_1, \ldots, a_n]$. However, computer scientists always start counting at 0 and therefore write it as $[a_0, \ldots, a_{n-1}]$. We use the computer science numbering here.