Chapter 2 (First-Order Logic) of

Verification of Object-Oriented Software

The KeY approach

1

First-Order Logic

by
Martin Giese

In this chapter, we introduce a first-order logic. This logic differs in some respects from what is introduced in most logic text books as classical first-order logic. The reason for the differences is that our logic has been designed in such a way that it is convenient for talking about JAVA programs. In particular our logic includes a type system with subtyping, a feature not found in most presentations of first-order logic.

Not only the logic itself, but also our presentation of it is different from what is found in a logic textbook: We concentrate on the definition of the language and its meaning, motivating most aspects from the intended application, namely the description of JAVA programs. We give many examples of the concepts introduced, to clarify what the definitions mean. In contrast to an ordinary logic text, we hardly state any theorems about the logic (with the notable exception of Section 1.6), and we prove none of them. The intention of this book is not to teach the reader how to *reason about* logics, but rather how to *use* one particular logic for a particular purpose.

The reader interested in the theoretical background of first-order logic in the context of automated deduction might want to read the book of Fitting [1996], or that of Goubault-Larrecq and Mackie [1997]. There are a number of textbooks covering first-order logic in general: by Ben-Ari [2003], Enderton [2000], Huth and Ryan [2004], Nerode and Shore [1979], or for the mathematically oriented reader Ebbinghaus et al. [1984]. To the best of our knowledge the only textbooks covering many-sorted logic, but not allowing subsorts, are those by Manzano [1996] and Gallier [1986]. For the technical details of the particular logic described in this chapter, see [Giese, 2005].

1.1 Types

We want to define the type system of our logic in a way that makes the logic particularly convenient to reason about objects of the JAVA programming

language. The type system of the logic therefore matches JAVA's type system in many ways.[1]

Before we define our type system, let us point out an important fact about the concept of types in JAVA.

In JAVA, there are two type concepts that should not be confused:

1. Every object created during the execution of a JAVA program has a *dynamic type*. If an object is created with the expression `new C(...)`, then `C` is the dynamic type of the newly created object. The dynamic type of an object is fixed from its creation until it is garbage collected. The dynamic type of an object can never be an interface type or an abstract class type.
2. Every expression occurring in a JAVA program has a *static type*. This static type is computed by the compiler from the literals, variables, methods, attributes, etc. that constitute the expression, using the type information in the declarations of these constituents. The static type is used for instance to determine which declaration an identifier refers to. A variable declaration `C x;` determines the static type `C` of the variable `x` when it occurs in an expression. Via a set of assignment compatibility rules, it also determines which static types are allowed for an expression `e` in an assignment `x = e`. In contrast to dynamic types, static types can also be abstract class types or interface types.

Every possible dynamic type can also occur as a static type. The static types are ordered in a type hierarchy. It therefore makes sense to talk about the dynamic type of an object being a subtype of some static type.

The connection between dynamic types and static types is this: The dynamic type of an object that results from evaluating an expression is always a subtype of the static type of that expression. For variables or attributes declared to be of type C, this means that the dynamic type of their value at runtime is always a subtype of C.

So, does a JAVA object have several types? No, an object has only a dynamic type, and it has exactly one dynamic type. However, an object can be used wherever a static type is required that is a supertype of its dynamic type.

We reflect this distinction in our logic by assigning static types to expressions ("terms") and dynamic types to their values ("domain elements").

We keep the discussion of the logic independent of any particular class library, by introducing the notion of a *type hierarchy*, which groups all the relevant information about the types and their subtyping relationships.

Definition 1.1. *A* type hierarchy *is a quadruple* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *of*

- *a finite set of* static types $\mathcal{T}$*,*
- *a finite set of* dynamic types $\mathcal{T}_d$*,*
- *a finite set of* abstract types $\mathcal{T}_a$*, and*

[1] It turns out that the resulting logic is reminiscent of Order-Sorted Algebras [Goguen and Meseguer, 1992].

- *a* subtype relation $\sqsubseteq$ *on* $\mathcal{T}$*,*

such that

- $\mathcal{T} = \mathcal{T}_d \dot{\cup} \mathcal{T}_a$
- *There is an* empty type $\bot \in \mathcal{T}_a$ *and a* universal type $\top \in \mathcal{T}_d$*.*
- $\sqsubseteq$ *is a reflexive partial order on* $\mathcal{T}$*, i.e., for all types* $A, B, C \in \mathcal{T}$*,*

$$A \sqsubseteq A$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq A \text{ then } A = B$$
$$\text{if } A \sqsubseteq B \text{ and } B \sqsubseteq C \text{ then } A \sqsubseteq C$$

- $\bot \sqsubseteq A \sqsubseteq \top$ *for all* $A \in \mathcal{T}$*.*
- $\mathcal{T}$ *is closed under greatest lower bounds w.r.t.* $\sqsubseteq$*, i.e., for any* $A, B \in \mathcal{T}$*, there is an*[2] $I \in \mathcal{T}$ *such that* $I \sqsubseteq A$ *and* $I \sqsubseteq B$ *and for any* $C \in \mathcal{T}$ *such that* $C \sqsubseteq A$ *and* $C \sqsubseteq B$*, it holds that* $C \sqsubseteq I$*. We write* $A \sqcap B$ *for the greatest lower bound of* A *and* B *and call it the* intersection type *of* A *and* B*. The existence of* $A \sqcap B$ *also guarantees the existence of the least upper bound* $A \sqcup B$ *of* A *and* B*, called the* union type *of* A *and* B*.*
- *Every non-empty abstract type* $A \in \mathcal{T}_a \setminus \{\bot\}$ *has a non-abstract subtype:* $B \in \mathcal{T}_d$ *with* $B \sqsubseteq A$*.*

We say that A is a subtype of B *if* $A \sqsubseteq B$*. The set of non-empty static types is denoted by* $\mathcal{T}_q := \mathcal{T} \setminus \{\bot\}$*.*

Note 1.2. In Java, interface types and abstract class types cannot be instantiated: the dynamic type of an object can never be an interface type or an abstract class type. We reflect this in our logic by dividing the set of types into two partitions:

$$\mathcal{T} = \mathcal{T}_d \dot{\cup} \mathcal{T}_a$$

$\mathcal{T}_d$ is the set of possible dynamic types, while $\mathcal{T}_a$ contains the abstract types, that can only occur as static types. The distinction between abstract class types and interface types is not important in this chapter, so we simply call all types that cannot be instantiated abstract.

The empty type $\bot$ is obviously abstract. Moreover, any abstract type that has no subtypes but $\bot$ would necessarily also be empty, so we require some non-abstract type to lie between any non-empty abstract type and the empty type.

Note 1.3. We consider only finite type hierarchies. In practice, any given Java program is finite, and can thus mention only finitely many types. The language specification actually defines infinitely many built-in types, namely the nested array types, e.g., `int[]`, `int[][]`, `int[][][]`, etc. Still, even though there are conceptually infinitely many types, any reasoning in our system is always in the context of a given fixed program, and only finitely many types are needed in that program.

[2] It is well-known that the greatest lower bound is unique if it exists.

The reason for restricting the logic to finite type hierarchies is that the construction of a calculus (⇒ Sect. 1.5) becomes problematic in the presence of infinite hierarchies and abstract types. We do not go into the intricate details in this text.

Note 1.4. We do not consider the universal type $\top$ to be abstract, which means that there might be objects that belong to $\top$, but to none of the more specific types. In JAVA this cannot happen: Any value is either of a primitive type or of a reference type, in which case its type is a subtype of `Object`. We can easily forbid objects of dynamic type $\top$ when we apply our logic to JAVA verification. On the other hand, simple explanatory examples that do not require a "real" type hierarchy are more easily formulated if $\top$ and $\bot$ are the only types.

Note 1.5. In JAVA, the primitive types `int`, `boolean`, etc. are conceptually quite different from the class and interface types. We do not need to make this difference explicit in our definition of the logic, at least not until a much later point. For the time being, the important property of an `int` value is that there are indeed values that have the type `int` and no other type at runtime. Hence, `int`, like all other primitive types, belongs to the dynamic, i.e., the non-abstract types.

Most of the notions defined in the remainder of this chapter depend on some type hierarchy. In order to avoid cluttering the notation, we assume that a certain fixed type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ is given, to which all later definitions refer.

Example 1.6. Consider the type hierarchy in Fig. 1.1, which is mostly taken from the JAVA Collections Framework. Arrows go from subtypes to supertypes, and abstract types are written in italic letters ($\bot$ is of course also abstract).

In this hierarchy, the following hold:

$$\mathcal{T} = \{\top, \text{Object}, \text{AbstractCollection}, \text{List}, \text{AbstractList}, \text{ArrayList}, \text{Null}, \text{int}, \bot\}$$

$$\mathcal{T}_q = \{\top, \text{Object}, \text{AbstractCollection}, \text{List}, \text{AbstractList}, \text{ArrayList}, \text{Null}, \text{int}\}$$

$$\mathcal{T}_d = \{\top, \text{Object}, \text{ArrayList}, \text{Null}, \text{int}\}$$

$$\mathcal{T}_a = \{\text{AbstractCollection}, \text{List}, \text{AbstractList}, \bot\}$$

$$\text{int} \sqcap \text{Object} = \bot$$

$$\text{int} \sqcup \text{Object} = \top$$

$$\text{AbstractCollection} \sqcap \text{List} = \text{AbstractList}$$

$$\text{AbstractCollection} \sqcup \text{List} = \text{Object}$$

$$\text{Object} \sqcap \text{Null} = \text{Null}$$

$$\text{Object} \sqcup \text{Null} = \text{Object}$$

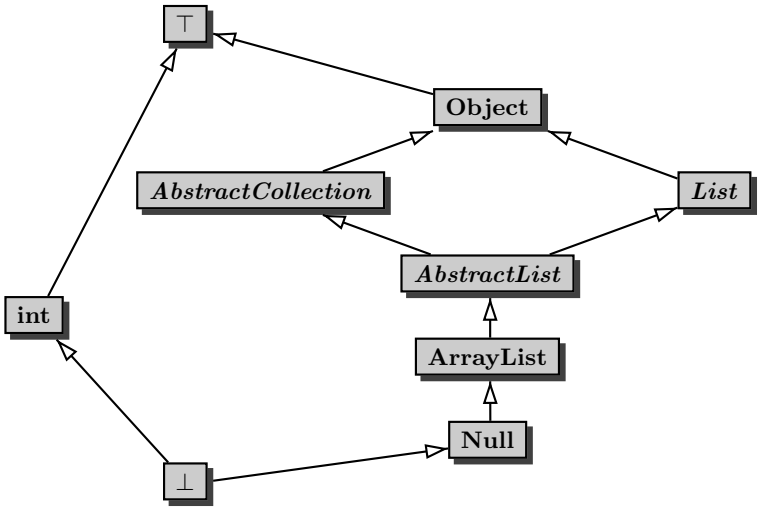

Fig. 1.1. An example type hierarchy

Example 1.7. Consider the type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ with:

$$\mathcal{T} := \{\top, \bot\}, \quad \mathcal{T}_d := \{\top\}, \quad \mathcal{T}_a := \{\bot\}, \quad \bot \sqsubseteq \top \ .$$

We call this the *minimal type hierarchy*. With this hierarchy, our notions are exactly like those for untyped first-order logic as introduced in other textbooks.

1.2 Signatures

A method in the Java programming language can be called, usually with a number of arguments, and it will in general compute a result which it returns. The same idea is present in the form of function or procedure definitions in many other programming languages.

The equivalent concepts in a logic are functions and predicates. A function gives a value depending on a number of arguments. A predicate is either true of false, depending on its arguments. In other words, a predicate is essentially a Boolean-valued function. But it is customary to consider functions and predicates separately.

In Java, every method has a declaration which states its name, the (static) types of the arguments it expects, the (static) type of its return value, and also other information like thrown exceptions, static or final flags, etc. The compiler uses the information in the declaration to determine whether it is

legal to call the method with a given list of arguments.[3] All types named in a declaration are static types. At run-time, the dynamic type of any argument may be a subtype of the declared argument type, and the dynamic type of the value returned may also be a subtype of the declared return type.

In our logic, we also fix the static types for the arguments of predicates and functions, as well as the return type of functions. The static types of all variables are also fixed. We call a set of such declarations a *signature*.

The main aspect of JAVA we want to reflect in our logic is its type system. Two constituents of JAVA expressions are particularly tightly linked to the meaning of dynamic and static types: type casts and `instanceof` expressions. A type cast `(A)o` changes the static type of an expression `o`, leaving the value (and therefore the dynamic type) unchanged. The expression `o instanceof A` checks whether the dynamic type of `o` is a subtype of `A`. There are corresponding operations in our logic. But instead of considering them to be special syntactic entities, we treat them like usual function resp. predicate symbols which we require to be present in any signature.

Definition 1.8. *A* signature *(for a given type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*) is a quadruple* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$ *of*

- *a set of* set of variable symbols VSym*,*
- *a set of* function symbols FSym*,*
- *a set of* predicate symbols PSym*, and*
- *a typing function* α*,*

such that[4]

- $\alpha(v) \in \mathcal{T}_q$ *for all* $v \in \mathrm{VSym}$*,*
- $\alpha(f) \in \mathcal{T}_q^* \times \mathcal{T}_q$ *for all* $f \in \mathrm{FSym}$*, and*
- $\alpha(p) \in \mathcal{T}_q^*$ *for all* $p \in \mathrm{PSym}$*.*
- *There is a function symbol* $(A) \in \mathrm{FSym}$ *with* $\alpha((A)) = ((\top), A)$ *for any* $A \in \mathcal{T}_q$*, called the* cast to type A*.*
- *There is a predicate symbol* $\doteq \in \mathrm{PSym}$ *with* $\alpha(\doteq) = (\top, \top)$*.*
- *There is a predicate symbol* $\sqsubseteq A \in \mathrm{PSym}$ *with* $\alpha(\sqsubseteq A) = (\top)$ *for any* $A \in \mathcal{T}$*, called the* type predicate for type A*.*

We use the following notations:

- $v{:}A$ *for* $\alpha(v) = A$*,*
- $f : A_1, \ldots, A_n \to A$ *for* $\alpha(f) = ((A_1, \ldots, A_n), A)$*, and*
- $p : A_1, \ldots, A_n$ *for* $\alpha(p) = (A_1, \ldots, A_n)$*.*

A constant symbol *is a function symbol* c *with* $\alpha(c) = ((), A)$ *for some type* A*.*

[3] The information is also used to disambiguate calls to overloaded methods, but this is not important here.

[4] We use the standard notation A^* to denote the set of (possibly empty) *sequences* of elements of A.

Note 1.9. We require the static types in signatures to be from $\mathcal{T}_q$, which excludes the empty type $\bot$. Declaring, for instance, a variable of the empty type would not be very sensible, since it would mean that the variable may not have any value. In contrast to JAVA, we allow using the Null type in a declaration, since it has the one element null.

Note 1.10. While the syntax $(A)t$ for type casts is the same as in JAVA, we use the syntax $t \sqsubseteq A$ instead of `instanceof` for type predicates. One reason for this is to save space. But the main reason is to remind ourselves that our type predicates have a slightly different semantics from that of the JAVA construct, as we will see in the following sections.

Note 1.11. In JAVA, there are certain restrictions on type casts: a cast to some type can only be applied to expressions of certain other types, otherwise the compiler signals an error. We are less restrictive in this respect, an object of any type may be cast to an object of any other (non-$\bot$) type. A similar observation holds for the type predicates, which may be applied in any situation, whereas JAVA's `instanceof` is subject to certain restrictions.

Note 1.12. We use the symbol $\doteq$ in our logic, to distinguish it from the equality $=$ of the mathematical meta-level. For instance, $t_1 \doteq t_2$ is a formula, while $\phi = (t_1 \doteq t_2)$ is a statement that two formulae are equal.

Like casts, our equality predicate $\doteq$ can be applied to terms of arbitrary types. It should be noted that the KeY system recognises certain cases where the equality is guaranteed not to hold and treats them as syntax errors. In particular, this happens for equalities between different primitive types and between a primitive type and a reference type. In contrast, the JAVA Language Specification also forbids equality between certain pairs of reference types. Both our logic and the implementation in the KeY system allow equalities between arbitrary reference types.

Note 1.13. In our discussion of the logic, we do not allow *overloading*: α gives a unique type to every symbol. This is not a real restriction: instead of an overloaded function f with $f : A \to B$ and $f : C \to D$, one can instead use two functions $f_1 : A \to B$ and $f_2 : C \to D$. Of course, the KeY system allows using overloaded methods in JAVA programs, but these are not represented as overloaded functions in the logic.

Example 1.14. For the type hierarchy from Example 1.6, see Fig. 1.1, a signature may contain:

$$\text{VSym} = \{n, o, l, a\} \quad \text{with} \quad n\text{:int}, \quad o\text{:Object}, \quad l\text{:List}, \quad a\text{:ArrayList}$$
$$\text{FSym} = \{\mathit{zero}, \mathit{plus}, \mathit{empty}, \mathit{length}, (\top), (\text{Object}), (\text{int}), \ldots\}$$

with

$$\begin{array}{ll} \mathit{zero}: & \mathrm{int} \\ \mathit{plus}: & \mathrm{int}, \mathrm{int} \to \mathrm{int} \\ \mathit{empty}: & \mathrm{List} \\ \mathit{length}: & \mathrm{List} \to \mathrm{int} \\ (\top): & \top \to \top \\ (\mathrm{Object}): & \top \to \mathrm{Object} \\ (\mathrm{int}): & \top \to \mathrm{int} \\ \quad \vdots & \end{array}$$

and

$$\mathrm{PSym} = \{\mathit{isEmpty}, \doteq, \sqsubseteq\top, \sqsubseteq\mathrm{Object}, \sqsubseteq\mathrm{int}, \ldots\}$$

with

$$\begin{array}{ll} \mathit{isEmpty}: & \mathrm{List} \\ \doteq: & \top, \top \\ \sqsubseteq\top: & \top \\ \sqsubseteq\mathrm{Object}: & \top \\ \sqsubseteq\mathrm{int}: & \top \\ \quad \vdots & \end{array}$$

In this example, *zero* and *empty* are constant symbols.

1.3 Terms and Formulae

Where the Java programming language has expressions, a logic has *terms* and *formulae*. Terms are composed by applying function symbols to variable and constant symbols.

Definition 1.15. *Given a signature* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$*, we inductively define the system of sets* $\{\mathrm{Trm}_A\}_{A \in \mathcal{T}}$ *of* terms of static type A *to be the least system of sets such that*

- $x \in \mathrm{Trm}_A$ *for any variable* $x{:}A \in \mathrm{VSym}$*,*
- $f(t_1, \ldots, t_n) \in \mathrm{Trm}_A$ *for any function symbol* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$*, and terms* $t_i \in \mathrm{Trm}_{A_i'}$ *with* $A_i' \sqsubseteq A_i$ *for* $i = 1, \ldots, n$*.*

For type cast terms, we write $(A)t$ *instead of* $(A)(t)$*. We write the static type of* t *as* $\sigma(t) := A$ *for any term* $t \in \mathrm{Trm}_A$*.*

A ground term *is a term that does not contain variables.*

Defining terms as the "least system of sets" with this property is just the mathematically precise way of saying that all entities built in the described way are terms, and *no others*.

Example 1.16. With the signature from Example 1.14, the following are terms:

n	a variable
empty	a constant
$\mathit{plus}(n, n)$	a function applied to two subterms
$\mathit{plus}(n, \mathit{plus}(n, n))$	nested function applications
$\mathit{length}(a)$	a function applied to a term of some subtype
$\mathit{length}((\text{List})o)$	a term with a type cast
$(\text{int})o$	a type cast we do not expect to "succeed"

On the other hand, the following are not terms:

$\mathit{plus}(n)$	wrong number of arguments
$\mathit{length}(o)$	wrong type of argument
$\mathit{isEmpty}(a)$	*isEmpty* is a predicate symbol, not a function symbol
$(\bot)n$	a cast to the empty type

Formulae are essentially Boolean-valued terms. They may be composed by applying predicate symbols to terms, but there are also some other ways of constructing formulae. Like with predicate and function symbols, the separation between terms and formulae in logic is more of a convention than a necessity. If one wants to draw a parallel to natural language, one can say that the formulae of a logic correspond to statements in natural language, while the terms correspond to the objects that the statements are about.

Definition 1.17. *We inductively define the set of formulae* Fml *to be the least set such that*

- $p(t_1, \ldots, t_n) \in \text{Fml}$ *for any predicate symbol* $p : A_1, \ldots, A_n$ *and terms* $t_i \in \text{Trm}_{A_i'}$ *with* $A_i' \sqsubseteq A_i$ *for* $i = 1, \ldots, n$,
- $\text{true}, \text{false} \in \text{Fml}$.
- $!\,\phi$, $(\phi \mid \psi)$, $(\phi \,\&\, \psi), (\phi \mathrel{->} \psi) \in \text{Fml}$ *for any* $\phi, \psi \in \text{Fml}$.
- $\forall x.\phi$, $\exists x.\phi \in \text{Fml}$ *for any* $\phi \in \text{Fml}$ *and any variable* x.

For type predicate formulae, we write $t \sqsubseteq A$ *instead of* $\sqsubseteq A(t)$. *For equalities, we write* $t_1 \doteq t_2$ *instead of* $\doteq(t_1, t_2)$. *An* atomic formula *or* atom *is a formula of the shape* $p(t_1, \ldots, t_n)$ *(including* $t_1 \doteq t_2$ *and* $t \sqsubseteq A$*). A* literal *is an atom or a negated atom* $!\,p(t_1, \ldots, t_n)$.

We use parentheses to disambiguate formulae. For instance, $(\phi \,\&\, \psi) \mid \xi$ and $\phi \,\&\, (\psi \mid \xi)$ are different formulae.

The intended meaning of the formulae is as follows:

$p(\ldots)$	The property p holds for the given arguments.
$t_1 \doteq t_2$	The values of t_1 and t_2 are equal.
true	always holds.
false	never holds.
$!\,\phi$	The formula ϕ does not hold.
$\phi \,\&\, \psi$	The formulae ϕ and ψ both hold.

$\phi \mid \psi$	At least one of the formulae ϕ and ψ holds.
ϕ –> ψ	If ϕ holds, then ψ holds.
$\forall x.\phi$	The formulae ϕ holds for all values of x.
$\exists x.\phi$	The formulae ϕ holds for at least one value of x.

In the next section, we give rigorous definitions that formalise these intended meanings.

KeY System Syntax, Textbook Syntax

The syntax used in this chapter is not exactly that used in the KeY system, mainly to save space and to make formulae easier to read. It is also different from the syntax used in other accounts of first-order logic, because that would make our syntax too different from the ASCII-oriented one actually used in the system. Below, we give the correspondence between the syntax of this chapter, that of the KeY system, and that of a typical introduction to first-order logic.

this chapter	KeY system	logic textbooks
$(A)t$	`(A)` t	—
$t \sqsubseteq A$	`A::contains(`t`)`	—
$t_1 \doteq t_2$	t_1 `=` t_2	$t_1 \doteq t_2$, $t_1 \approx t_2$, etc.
true	`true`	T, tt, $\top$, etc.
false	`false`	F, ff, $\bot$, etc.
$!\phi$	`!` ϕ	$\neg\phi$
ϕ & ψ	ϕ `&` ψ	$\phi \wedge \psi$
$\phi \mid \psi$	ϕ `|` ψ	$\phi \vee \psi$
ϕ –> ψ	ϕ `->` ψ	$\phi \rightarrow \psi$
$\forall x.\phi$	`\forall A x;` ϕ	$\forall x.\phi$, $(\forall x)\phi$, etc.
$\exists x.\phi$	`\exists A x;` ϕ	$\exists x.\phi$, $(\exists x)\phi$, etc.

The KeY system requires the user to give a type for the bound variable in quantifiers. In fact, the system does not know of a global set VSym of variable symbols with a fixed typing function α, as we suggested in Def. 1.8. Instead, each variable is "declared" by the quantifier that binds it, so that is also where the type is given.

Concerning the "conventional" logical syntax, note that most accounts of first-order logic do not discuss subtypes, and accordingly, there is no need for type casts or type predicates. Also note that the syntax can vary considerably, even between conventional logic textbooks.

The operators $\forall$ and $\exists$ are called the *universal* and *existential quantifier*, respectively. We say that they *bind* the variable x in the sub-formula ϕ, or that ϕ is the *scope* of the quantified variable x. This is very similar to the way in which a JAVA method body is the scope of the formal parameters declared in the method header. All variables occurring in a formula that are *not* bound

by a quantifier are called *free*. For the calculus that is introduced later in this chapter, we are particularly interested in *closed* formulae, which have no free variables. These intuitions are captured by the following definition:

Definition 1.18. *We define $fv(t)$, the set of free variables of a term t, by*

- $fv(v) = \{v\}$ *for* $v \in \mathrm{VSym}$, *and*
- $fv(f(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$.

The set of free variables of a formula is defined by

- $fv(p(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$,
- $fv(t_1 \doteq t_2) = fv(t_1) \cup fv(t_2)$,
- $fv(\mathrm{true}) = fv(\mathrm{false}) = \emptyset$,
- $fv(!\,\phi) = fv(\phi)$,
- $fv(\phi \;\&\; \psi) = fv(\phi \mid \psi) = fv(\phi \mathrel{->} \psi) = fv(\phi) \cup fv(\psi)$, *and*
- $fv(\forall x.\phi) = fv(\exists x.\phi) = fv(\phi) \setminus \{x\}$.

A formula ϕ is called closed *iff $fv(\phi) = \emptyset$.*

Example 1.19. Given the signature from Example 1.14, the following are formulae:

$isEmpty(a)$	an atomic formula with free variable a
$a \doteq empty$	an equality atom with free variable a
$o \sqsubseteq \mathrm{List}$	a type predicate atom with free variable o
$o \sqsubseteq \bot$	a type predicate atom for the empty type with free variable o
$\forall l.(length(l) \doteq zero \mathrel{->} isEmpty(l))$	a closed formula with a quantifier
$o \doteq empty \mid \forall o.o \sqsubseteq \top$	a formula with one free and one bound occurrence of o

On the other hand, the following are not formulae:

$length(l)$	*length* is not a predicate symbol.
$isEmpty(o)$	wrong argument type
$isEmpty(isEmpty(a))$	applying predicate on formula, instead of term
$a = empty$	equality should be $\doteq$
$\forall l.length(l)$	applying a quantifier to a term

1.4 Semantics

So far, we have only discussed the syntax, the textual structure of our logic. The next step is to assign a meaning, known as a *semantics*, to the terms and formulae.

1.4.1 Models

For compound formulae involving &, |, ∀, etc., our definition of a semantics should obviously correspond to their intuitive meaning as explained in the previous section. What is not clear is how to assign a meaning in the "base case", i.e., what is the meaning of atomic formulae like $p(a)$. It seems clear that this should depend on the meaning of the involved terms, so the semantics of terms also needs to be defined.

We do this by introducing the concept of a *model*. A model assigns a meaning (in terms of mathematical entities) to the basic building blocks of our logic, i.e., the types, and the function and predicate symbols. We can then define how to combine these meanings to obtain the meaning of any term or formula, always with respect to some model.

Actually, a model fixes only the meaning of function and predicate symbols. The meaning of the third basic building block, namely the variables is given by *variable assignments* which is introduced in Def. 1.23.[5]

When we think of a method call in a JAVA program, the returned value depends not only on the values of the arguments, but possibly also on the state of some other objects. Calling the method again in a modified state might give a different result. In this chapter, we do not take into account this idea of a changing state. A model gives a meaning to any term or formula, and in the same model, this meaning never changes. Evolving states will become important in Chapter **??**.

Before we state the definition, let us look into type casts, which receive a special treatment. Recall that in JAVA, the evaluation of a type cast expression `(A)o` checks whether the value of `o` has a dynamic type equal to `A` or a subtype of `A`. If this is the case, the value of the cast is the same as the value of `o`, though the expression `(A)o` has static type `A`, independently of what the static type of `o` was. If the dynamic type of the value of `o` does *not* fit the type `A`, a `ClassCastException` is thrown.

In a logic, we want *every* term to have a *value*. It would greatly complicate things if we had to take things like exceptions into account. We therefore take the following approach:

1. The value of a term $(A)t$ is the same as the value of t, provided the value of t "fits" the type A.
2. Otherwise, the term is given an arbitrary value, but still one that "fits" its static type A.

If we want to differentiate between these two cases, we can use a type predicate formula $t \sqsubseteq A$: this is defined to hold exactly if the value of t "fits" the type A.

Definition 1.20. *Given a type hierarchy and a signature as before, a* model *is a triple* $(\mathcal{D}, \delta, \mathcal{I})$ *of*

[5] The reason for keeping the variables separate is that the variable assignment is occasionally modified in the semantic definitions, whereas the model stays the same.

- *a* domain $\mathcal{D}$*,*
- *a* dynamic type function $\delta : \mathcal{D} \to \mathcal{T}_d$*, and*
- *an* interpretation $\mathcal{I}$*,*

such that, if we define[6]

$$\mathcal{D}^A := \{d \in \mathcal{D} \mid \delta(d) \sqsubseteq A\} \ ,$$

it holds that

- $\mathcal{D}^A$ *is non-empty for all* $A \in \mathcal{T}_d$*,*
- *for any* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$*,* $\mathcal{I}$ *yields a function*

$$\mathcal{I}(f) : \mathcal{D}^{A_1} \times \ldots \times \mathcal{D}^{A_n} \to \mathcal{D}^A \ ,$$

- *for any* $p : A_1, \ldots, A_n \in \mathrm{PSym}$*,* $\mathcal{I}$ *yields a subset*

$$\mathcal{I}(p) \subseteq \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n} \ ,$$

- *for type casts,* $\mathcal{I}((A))(x) = x$ *if* $\delta(x) \sqsubseteq A$*, otherwise* $\mathcal{I}((A))(x)$ *is an arbitrary but fixed*[7] *element of* $\mathcal{D}^A$*, and*
- *for equality,* $\mathcal{I}(\doteq) = \{(d, d) \mid d \in \mathcal{D}\}$*,*
- *for type predicates,* $\mathcal{I}(\sqsubseteq A) = \mathcal{D}^A$*.*

As we promised in the beginning of Section 1.1, every domain element d has a dynamic type $\delta(d)$, just like every object created when executing a Java program has a dynamic type. Also, just like in Java, the dynamic type of a domain element cannot be an abstract type.

Example 1.21. For the type hierarchy from Example 1.6 and the signature from Example 1.14, the "intended" model $\mathcal{M}_1 = (\mathcal{D}, \delta, \mathcal{I})$ may be described as follows:

Given a state in the execution of a Java program, let AL be the set of all existing `ArrayList` objects. We assume that there is at least one `ArrayList` object `e` that is currently empty. We denote some arbitrary but fixed `ArrayList` object (possibly equal to `e`) by `o`. Also, let $I := \{-2^{31}, \ldots, 2^{31} - 1\}$ be the set of all possible values for a Java `int`.[8] Now let

$$\mathcal{D} := AL \,\dot\cup\, I \,\dot\cup\, \{\texttt{null}\} \ .$$

We define δ by

[6] $\mathcal{D}^A$ is our formal definition of the set of all domain elements that "fit" the type A.

[7] The chosen element may be different for different arguments, i.e., if $x \neq y$, then $\mathcal{I}((A))(x) \neq \mathcal{I}((A))(y)$ is allowed.

[8] The question of how best to reason about Java arithmetic is actually quite complex, and is covered in Chapter **??**. Here, we take a restricted range of integers for the purpose of explaining the concept of a model.

$$\delta(d) := \begin{cases} \text{int} & \text{if } d \in I \\ \text{ArrayList} & \text{if } d \in AL \\ \text{Null} & \text{if } d = \texttt{null} \end{cases}$$

With those definitions, we get

$$\begin{gathered} \mathcal{D}^{\top} = AL \,\dot\cup\, I \,\dot\cup\, \{\texttt{null}\} \\ \mathcal{D}^{\text{int}} = I \\ \mathcal{D}^{\text{Object}} = \mathcal{D}^{\text{AbstractCollection}} = \mathcal{D}^{\text{List}} = \\ \mathcal{D}^{\text{AbstractList}} = \mathcal{D}^{\text{ArrayList}} = AL \,\dot\cup\, \{\texttt{null}\} \\ \mathcal{D}^{\text{Null}} = \{\texttt{null}\} \\ \mathcal{D}^{\bot} = \emptyset \end{gathered}$$

Now, we can fix the interpretations of the function symbols:

$$\begin{aligned} \mathcal{I}(\mathit{zero})() &:= 0 \\ \mathcal{I}(\mathit{plus})(x, y) &:= x + y \quad \text{(with JAVA's overflow behaviour)} \\ \mathcal{I}(\mathit{empty})() &:= \texttt{e} \\ \mathcal{I}(\mathit{length})(l) &:= \begin{cases} \text{the length of } l & \text{if } l \neq \texttt{null} \\ 0 & \text{if } l = \texttt{null} \end{cases} \end{aligned}$$

Note that the choice of 0 for the *length* of `null` is arbitrary, since `null` does not represent a list. Most of the interpretation of casts is fixed, but it needs to be completed for arguments that are not of the "right" type:

$$\begin{aligned} \mathcal{I}((\top))(d) &:= d \\ \mathcal{I}((\text{int}))(d) &:= \begin{cases} d & \text{if } d \in I \\ 23 & \text{otherwise} \end{cases} \\ \mathcal{I}((\text{Object}))(d) &:= \begin{cases} d & \text{if } d \in AL \,\dot\cup\, \{\texttt{null}\} \\ \texttt{o} & \text{otherwise} \end{cases} \\ &\vdots \end{aligned}$$

Note how the interpretation must produce a value of the correct type for every combination of arguments, even those that would maybe lead to a `NullPointerException` or a `ClassCastException` in JAVA execution. For the *isEmpty* predicate, we can define:

$$\mathcal{I}(\mathit{isEmpty}) := \{l \in AL \mid l \text{ is an empty } \texttt{ArrayList}\} \quad .$$

The interpretation of $\doteq$ and of the type predicates is fixed by the definition of a model:

$$\begin{aligned} \mathcal{I}(\doteq) &:= \{(d, d) \mid d \in \mathcal{D}\} \\ \mathcal{I}(\sqsubseteq\top) &:= AL \,\dot\cup\, I \,\dot\cup\, \{\texttt{null}\} \\ \mathcal{I}(\sqsubseteq\text{int}) &:= I \\ \mathcal{I}(\sqsubseteq\text{Object}) &:= AL \,\dot\cup\, \{\texttt{null}\} \\ &\vdots \end{aligned}$$

Example 1.22. While the model in the previous example follows the intended meanings of the types, functions, and predicates quite closely, there are also models that have a completely different behaviour. For instance, we can define a model $\mathcal{M}_2$ with

$$\mathcal{D} := \{\square, \diamond\} \quad \text{with} \quad \delta(\square) := \text{int} \quad \text{and} \quad \delta(\diamond) := \text{Null} \ .$$

This gives us:

$$\begin{gathered} \mathcal{D}^{\top} = \{\square, \diamond\} \\ \mathcal{D}^{\text{int}} = \{\square\} \\ \mathcal{D}^{\text{Object}} = \mathcal{D}^{\text{AbstractCollection}} = \mathcal{D}^{\text{List}} = \\ \mathcal{D}^{\text{AbstractList}} = \mathcal{D}^{\text{ArrayList}} = \mathcal{D}^{\text{Null}} = \{\diamond\} \\ \mathcal{D}^{\bot} = \emptyset \end{gathered}$$

The interpretation of the functions can be given by:

$$\begin{aligned} \mathcal{I}(\mathit{zero})() &:= \square & \mathcal{I}((\top))(d) &:= d \\ \mathcal{I}(\mathit{plus})(x, y) &:= \square & \mathcal{I}((\text{int}))(d) &:= \square \\ \mathcal{I}(\mathit{empty})() &:= \diamond & \mathcal{I}((\text{Object}))(d) &:= \diamond \\ \mathcal{I}(\mathit{length})(l) &:= \square & & \vdots \end{aligned}$$

and the predicates by:

$$\begin{aligned} \mathcal{I}(\mathit{isEmpty}) &:= \emptyset \\ \mathcal{I}(\doteq) &:= \{(\square, \square), (\diamond, \diamond)\} \\ \mathcal{I}(\sqsubseteq \top) &:= \{\square, \diamond\} \\ \mathcal{I}(\sqsubseteq \text{int}) &:= \{\square\} \\ \mathcal{I}(\sqsubseteq \text{Object}) &:= \{\diamond\} \\ &\vdots \end{aligned}$$

The following definitions apply to this rather nonsensical model as well as to the one defined in the previous example. In Section 1.4.3, we introduce a way of restricting which models we are interested in.

1.4.2 The Meaning of Terms and Formulae

A model is not quite sufficient to give a meaning to an arbitrary term or formula: it says nothing about the variables. For this, we introduce the notion of a variable assignment.

Definition 1.23. *Given a model* $(\mathcal{D}, \delta, \mathcal{I})$*, a* variable assignment *is a function* $\beta : \text{VSym} \to \mathcal{D}$*, such that*

$$\beta(x) \in \mathcal{D}^A \quad \textit{for all} \quad x{:}A \in \text{VSym} \ .$$

We also define the modification β_x^d *of a variable assignment* β *for any variable* $x{:}A$ *and any domain element* $d \in \mathcal{D}^A$ *by:*

$$\beta_x^d(y) := \begin{cases} d & \text{if } y = x \\ \beta(y) & \text{otherwise} \end{cases}$$

We are now ready to define the semantics of terms.

Definition 1.24. *Let* $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ *be a model, and* β *a variable assignment. We inductively define the valuation function* $\mathrm{val}_{\mathcal{M}}$ *by*

- $\mathrm{val}_{\mathcal{M},\beta}(x) = \beta(x)$ *for any variable* x.
- $\mathrm{val}_{\mathcal{M},\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_{\mathcal{M},\beta}(t_1), \ldots, \mathrm{val}_{\mathcal{M},\beta}(t_n))$.

For a ground term t*, we simply write* $\mathrm{val}_{\mathcal{M}}(t)$*, since* $\mathrm{val}_{\mathcal{M},\beta}(t)$ *is independent of* β.

Example 1.25. Given the signature from Example 1.14 and the models $\mathcal{M}_1$ and $\mathcal{M}_2$ from Examples 1.21 and 1.22, we can define variable assignments β_1 resp. β_2 as follows:

$\beta_1(n) := 5$	$\beta_2(n) := \square$
$\beta_1(o) := \texttt{null}$	$\beta_2(o) := \diamond$
$\beta_1(l) := \texttt{e}$	$\beta_2(l) := \diamond$
$\beta_1(a) := \texttt{e}$	$\beta_2(a) := \diamond$

We then get the following values for the terms from Example 1.16:

t	$\mathrm{val}_{\mathcal{M}_1,\beta_1}(t)$	$\mathrm{val}_{\mathcal{M}_2,\beta_2}(t)$
n	5	$\square$
empty	e	$\diamond$
plus(n, n)	10	$\square$
plus$(n,$ *plus*$(n, n))$	15	$\square$
length(a)	0	$\square$
length$((\mathrm{List})o)$	0	$\square$
$(\mathrm{int})o$	23	$\square$

The semantics of formulae is defined in a similar way: we define a validity relation that says whether some formula is valid in a given model under some variable assignment.

Definition 1.26. *Let* $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ *be a model, and* β *a variable assignment. We inductively define the validity relation* $\models$ *by*

- $\mathcal{M}, \beta \models p(t_1, \ldots, t_n)$ *iff* $(\mathrm{val}_{\mathcal{M},\beta}(t_1), \ldots, \mathrm{val}_{\mathcal{M},\beta}(t_n)) \in \mathcal{I}(p)$.
- $\mathcal{M}, \beta \models \mathrm{true}$.
- $\mathcal{M}, \beta \not\models \mathrm{false}$.

- $\mathcal{M}, \beta \models !\,\phi$ *iff* $\mathcal{M}, \beta \not\models \phi$.
- $\mathcal{M}, \beta \models \phi$ & ψ *iff* $\mathcal{M}, \beta \models \phi$ *and* $\mathcal{M}, \beta \models \psi$.
- $\mathcal{M}, \beta \models \phi \mid \psi$ *iff* $\mathcal{M}, \beta \models \phi$ *or* $\mathcal{M}, \beta \models \psi$, *or both.*
- $\mathcal{M}, \beta \models \phi$ –> ψ *iff* if $\mathcal{M}, \beta \models \phi$, *then also* $\mathcal{M}, \beta \models \psi$.
- $\mathcal{M}, \beta \models \forall x.\phi$ *(for a variable* x:A*) iff* $\mathcal{M}, \beta^d_x \models \phi$ *for every* $d \in \mathcal{D}^A$.
- $\mathcal{M}, \beta \models \exists x.\phi$ *(for a variable* x:A*) iff there is some* $d \in \mathcal{D}^A$ *such that* $\mathcal{M}, \beta^d_x \models \phi$.

If $\mathcal{M}, \beta \models \phi$, *we say that* ϕ *is* valid in the model $\mathcal{M}$ under the variable assignment β. *For a closed formula* ϕ, *we write* $\mathcal{M} \models \phi$, *since* β *is then irrelevant.*

Example 1.27. Let us consider the semantics of the formula

$$\forall l.(\mathit{length}(l) \doteq \mathit{zero} \text{ –> } \mathit{isEmpty}(l))$$

in the model $\mathcal{M}_1$ described in Example 1.21. Intuitively, we reason as follows: the formula states that any list l which has length 0 is empty. But in our model, `null` is a possible value for l, and `null` has length 0, but is not considered an empty list. So the statement does not hold.

Formally, we start by looking at the smallest constituents and proceed by investigating the validity of larger and larger sub-formulae.

1. Consider the term $\mathit{length}(l)$. Its value $\mathrm{val}_{\mathcal{M}_1,\beta}(\mathit{length}(l))$ is the length of the `ArrayList` object identified by $\beta(l)$, or 0 if $\beta(l) = \texttt{null}$.
2. $\mathrm{val}_{\mathcal{M}_1,\beta}(\mathit{zero})$ is 0.
3. Therefore, $\mathcal{M}_1, \beta \models \mathit{length}(l) \doteq \mathit{zero}$ exactly if $\beta(l)$ is an `ArrayList` object of length 0, or $\beta(l)$ is `null`.
4. $\mathcal{M}_1, \beta \models \mathit{isEmpty}(l)$ iff $\beta(l)$ is an empty `ArrayList` object.
5. Whenever the length of an `ArrayList` object is 0, it is also empty.
6. `null` is *not* an empty `ArrayList` object.
7. Hence, $\mathcal{M}_1, \beta \models \mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l)$ holds iff $\beta(l)$ is not `null`.
8. For any β, we have $\mathcal{M}_1, \beta^{\texttt{null}}_l \not\models \mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l)$, because $\beta^{\texttt{null}}_l(l) = \texttt{null}$.
9. Therefore, $\mathcal{M}_1, \beta \not\models \forall l.(\mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l))$.

In the other model, $\mathcal{M}_2$ from Example 1.22,

1. $\mathrm{val}_{\mathcal{M}_2,\beta}(\mathit{length}(l)) = \square$, whatever $\beta(l)$ is.
2. $\mathrm{val}_{\mathcal{M}_2,\beta}(\mathit{zero})$ is also $\square$.
3. Therefore, $\mathcal{M}_2, \beta \models \mathit{length}(l) \doteq \mathit{zero}$ holds for any β.
4. There is *no* $\beta(l)$ such that $\mathcal{M}_2, \beta \models \mathit{isEmpty}(l)$ holds.
5. Thus, there is no β such that $\mathcal{M}_2, \beta \models \mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l)$.
6. In particular, $\mathcal{M}_2, \beta^{\texttt{null}}_l \not\models \mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l)$ for all β.
7. Therefore, $\mathcal{M}_2, \beta \not\models \forall l.(\mathit{length}(l) \doteq \mathit{zero}$ –> $\mathit{isEmpty}(l))$.

This result is harder to explain intuitively, since the model $\mathcal{M}_2$ is itself unintuitive. But our description of the model and the definitions of the semantics allow us to determine the truth of any formula in the model.

In the example, we have seen a formula that is valid in neither of the two considered models. However, the reader might want to check that there are also models in which the formula holds.[9] But there are also formulae that hold in all models, or in none. We have special names for such formulae.

Definition 1.28. *Let a fixed type hierarchy and signature be given.*[10]

- *A formula ϕ is* logically valid *if $\mathcal{M}, \beta \models \phi$ for any model $\mathcal{M}$ and any variable assignment β.*
- *A formula ϕ is* satisfiable *if $\mathcal{M}, \beta \models \phi$ for some model $\mathcal{M}$ and some variable assignment β.*
- *A formula is* unsatisfiable *if it is not satisfiable.*

It is important to realize that logical validity is a very different notion from the validity in a particular model. We have seen in our examples that there are many models for any given signature, most of them having nothing to do with the intended meaning of symbols. While validity in a model is a relation between a formula and a model (and a variable assignment), logical validity is a property of a formula. In Section 1.5, we show that it is even possible to check logical validity without ever talking about models.

For the time being, here are some examples where the validity/satisfiability of simple formulae is determined through explicit reasoning about models.

Example 1.29. For any formula ϕ, the formula

$$\phi \mid !\,\phi$$

is logically valid: Consider the semantics of ϕ. For any model $\mathcal{M}$ and any variable assignment β, either $\mathcal{M}, \beta \models \phi$, or not. If $\mathcal{M}, \beta \models \phi$, the semantics of $\mid$ in Def. 1.26 tells us that also $\mathcal{M}, \beta \models \phi \mid !\,\phi$. Otherwise, the semantics of the negation $!$ tells us that $\mathcal{M}, \beta \models !\,\phi$, and therefore again $\mathcal{M}, \beta \models \phi \mid !\,\phi$. So our formula holds in any model, under any variable assignment, and is thus logically valid.

Example 1.30. For any formula ϕ, the formula

$$\phi \;\&\; !\,\phi$$

is unsatisfiable: Consider an arbitrary, but fixed model $\mathcal{M}$ and a variable assignment β. For $\mathcal{M}, \beta \models \phi \;\&\; !\,\phi$ to hold, according to Def. 1.26, both $\mathcal{M}, \beta \models \phi$ and $\mathcal{M}, \beta \models !\,\phi$ must hold. This cannot be the case, because of the semantics of $!$. Hence, $\mathcal{M}, \beta \models \phi \;\&\; !\,\phi$ does not hold, irrespective of the model and the variable assignment, which means that the formula is unsatisfiable.

[9] Hint: define a model like $\mathcal{M}_1$, but let $\mathcal{I}(\mathit{length})(\texttt{null}) = -1$.

[10] It is important to fix the type hierarchy: there are formulae which are logically valid in some type hierarchies, unsatisfiable in others, and satisfiable but not valid in others still. For instance, it might amuse the interested reader to look for such type hierarchies for the formula $\exists x.x \sqsubseteq A \;\&\; !\,x \sqsubseteq B$.

Example 1.31. The formula

$$\exists x.x \doteq x$$

for some variable $x{:}A$ with $A \in \mathcal{T}_q$ is logically valid: Consider an arbitrary, but fixed model $\mathcal{M}$ and a variable assignment β. We have required in Def. 1.20 that $\mathcal{D}^A$ is non-empty. Pick an arbitrary element $a \in \mathcal{D}^A$ and look at the modified variable assignment β^a_x. Clearly, $\mathcal{M}, \beta^a_x \models x \doteq x$, since both sides of the equation are equal terms and must therefore evaluate to the same domain element (namely a). According to the semantics of the $\exists$ quantifier in Def. 1.26, it follows that $\mathcal{M}, \beta \models x \doteq x$. Since this holds for any model and variable assignment, the formula is logically valid.

Example 1.32. The formula

$$\forall l.(\mathit{length}(l) \doteq \mathit{zero} \mathrel{-\!\!>} \mathit{isEmpty}(l))$$

is satisfiable. It is not logically valid, since it does not hold in every model, as we have seen in Example 1.27. To see that it is satisfiable, take a model $\mathcal{M}$ with

$$\mathcal{I}(\mathit{isEmpty}) := \mathcal{D}^{\mathrm{List}}$$

so that $\mathit{isEmpty}(l)$ is true for every value of l. Accordingly, in $\mathcal{M}$, the implication $\mathit{length}(l) \doteq \mathit{zero} \mathrel{-\!\!>} \mathit{isEmpty}(l)$ is also valid for any variable assignment. The semantics of the $\forall$ quantifier then tells us that

$$\mathcal{M} \models \forall l.(\mathit{length}(l) \doteq \mathit{zero} \mathrel{-\!\!>} \mathit{isEmpty}(l))$$

so the formula is indeed satisfied by $\mathcal{M}$.

Example 1.33. The formula

$$(A)x \doteq x \mathrel{-\!\!>} x \sqsubseteq A$$

with $x{:}\top$ is logically valid for any type hierarchy and any type A: Remember that

$$\mathrm{val}_{\mathcal{M},\beta}((A)x) = \mathcal{I}((A))(\beta(x)) \in \mathcal{D}^A \ .$$

Now, if $\beta(x) \in \mathcal{D}^A$, then $\mathrm{val}_{\mathcal{M},\beta}((A)x) = \beta(x)$, so $\mathcal{M}, \beta \models (A)x \doteq x$. On the other hand, if $\beta(x) \notin \mathcal{D}^A$, then it cannot be equal to $\mathrm{val}_{\mathcal{M},\beta}((A)x)$, so $\mathcal{M}, \beta \not\models (A)x \doteq x$. Thus, if $(A)x \doteq x$, holds, then $\beta(x) \in \mathcal{D}^A$, and therefore $\mathcal{M}, \beta \models x \sqsubseteq A$.

The converse

$$x \sqsubseteq A \mathrel{-\!\!>} (A)x \doteq x$$

is also logically valid for any type hierarchy and any type A: if $\mathcal{M}, \beta \models x \sqsubseteq A$, then $\beta \in \mathcal{D}^A$, and therefore $\mathcal{M}, \beta \models (A)x \doteq x$.

Logical Consequence

A concept that is quite central to many other introductions to logic, but that is hardly encountered when dealing with the KeY system, is that of *logical consequence*. We briefly explain it here.

Given a set of closed formulae M and a formula ϕ, ϕ is said to be a *logical consequence* of M, written $M \models \phi$, iff for all models $\mathcal{M}$ and variable assignments β such that $\mathcal{M}, \beta \models \psi$ for all $\psi \in M$, it also holds that $\mathcal{M}, \beta \models \phi$.

In other words, ϕ is not required to be satisfied in all models and under all variable assignments, but only under those that satisfy all elements of M.

For instance, for any closed formulae ϕ and ψ, $\{\phi, \psi\} \models \phi \,\&\, \psi$, since $\phi \,\&\, \psi$ holds for all $\mathcal{M}, \beta$ for which both ϕ and ψ hold.

Two formulae ϕ and ψ are called *logically equivalent* if for all models $\mathcal{M}$ and variable assignments β, $\mathcal{M}, \beta \models \phi$ iff $\mathcal{M}, \beta \models \psi$.

Note 1.34. The previous example shows that type predicates are not really necessary in our logic, since a sub-formula $t \sqsubseteq A$ could always be replaced by $(A)t \doteq t$. In the terminology of the above sidebar, the two formulae are logically equivalent. Another formula that is easily seen to be logically equivalent to $t \sqsubseteq A$ is

$$\exists y.y \doteq t$$

with a variable $y{:}A$. It is shown in Section 1.5.6 however, that the main way of reasoning about types, and in particular about type casts in our calculus is to collect information about dynamic types using type predicates. Therefore, adding type predicates to our logic turns out to be the most convenient approach for reasoning, even if they do not add anything to the expressivity.

1.4.3 Partial Models

Classically, the logically valid formulae have been at the centre of attention when studying a logic. However, when dealing with formal methods, many of the involved types have a fixed *intended* meaning. For instance, in our examples, the type int is certainly intended to denote the 4 byte two's complement integers of the JAVA language, and the function symbol *plus* should denote the addition of such integers.[11] On the other hand, for some types and symbols, we *are* interested in all possible meanings.

To formally express this idea, we introduce the concept of a *partial model*, which gives a meaning to parts of a type hierarchy and signature. We then define what it means for a model to extend a partial model, and look only at such models.

[11] We repeat that the issue of reasoning about JAVA arithmetic in the KeY system is actually more complex (⇒ Chap. **??**).

The following definition of a partial model is somewhat more complex than might be expected. If we want to fix the interpretation of some of the functions and predicates in our signature, it is not sufficient to say which, and to give their interpretations. The interpretations must act on some domain, and the domain elements must have some type. For instance, if we want *plus* to represent the addition of JAVA integers, we must also identify a subset of the domain which should be the domain for the int type.

In addition, we want it to be possible to fix the interpretation of some functions only on parts of the domain. For instance, we might not want to fix the result of a division by zero.[12]

Definition 1.35. *Given a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *and a corresponding signature* $(\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym}, \alpha)$*, we define a* partial model *to be a quintuple* $(\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ *consisting of*

- *a set of* fixed types $\mathcal{T}_0 \subseteq \mathcal{T}_d$*,*
- *a set* $\mathcal{D}_0$ *called the* partial domain*,*
- *a* dynamic type function $\delta_0 : \mathcal{D}_0 \to \mathcal{T}_0$*,*
- *a* fixing function D_0*, and*
- *a* partial interpretation $\mathcal{I}_0$*,*

where

- $\mathcal{D}_0^A := \{d \in \mathcal{D}_0 \mid \delta_0(d) \sqsubseteq A\}$ *is non-empty for all* $A \in \mathcal{T}_0$*,*
- *for any* $f : A_1, \ldots, A_n \to A_0 \in \mathrm{FSym}$ *with all* $A_i \in \mathcal{T}_0$*,* D_0 *yields a set of tuples of domain elements*

$$D_0(f) \subseteq \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$$

and $\mathcal{I}_0$ *yields a function*

$$\mathcal{I}_0(f) : D_0(f) \to \mathcal{D}_0^{A_0} \ ,$$

and

- *for any* $p : A_1, \ldots, A_n \in \mathrm{PSym}$ *with all* $A_i \in \mathcal{T}_0$*,* D_0 *yields a set of tuples of domain elements*

$$D_0(p) \subseteq \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n}$$

and $\mathcal{I}_0$ *yields a subset*

$$\mathcal{I}_0(p) \subseteq D_0(p) \ ,$$

and

- *for any* $f : A_1, \ldots, A_n \to A_0 \in \mathrm{FSym}$*, resp.* $p : A_1, \ldots, A_n \in \mathrm{PSym}$ *with one of the* $A_i \notin \mathcal{T}_0$*,* $D_0(f) = \emptyset$*, resp.* $D_0(p) = \emptyset$*.*

[12] Instead of using *partial functions* for cases like division by zero, i.e., functions which do not have a value for certain arguments, we consider our functions to be total, but we might not fix (or know, or care about) the value for some arguments. This corresponds to the under-specification approach advocated by Hähnle [2005].

This is a somewhat complex definition, so we explain the meaning of its various parts. As mentioned above, a part of the domain needs to be fixed for the interpretation to act upon, and the dynamic type of each element of that partial domain needs to be identified. This is the role of $\mathcal{T}_0$, $\mathcal{D}_0$, and δ_0. The fixing function D_0 says for which tuples of domain elements and for which functions this partial model should prescribe an interpretation. In particular, if D_0 gives an empty set for some symbol, then the partial model does not say anything at all about the interpretation of that symbol. If D_0 gives the set of all element tuples corresponding to the signature of that symbol, then the interpretation of that symbol is completely fixed. Consider the special case of a constant symbol c: there is only one 0-tuple, namely $()$, so the fixing function can be either $D_0(c) = \{()\}$, meaning that the interpretation of c is fixed to some domain element $\mathcal{I}_0(c)()$, or $D_0(c) = \emptyset$, meaning that it is not fixed.

Finally, the partial interpretation $\mathcal{I}_0$ specifies the interpretation for those tuples of elements where the interpretation should be fixed.

Example 1.36. We use the type hierarchy from the previous examples, and add to the signature from Example 1.14 a function symbol $\mathit{div} : \mathrm{int}, \mathrm{int} \to \mathrm{int}$. We want to define a partial model that fixes the interpretation of *plus* to be the two's complement addition of four-byte integers that is used by Java. The interpretation of *div* should behave like Java's division unless the second argument is zero, in which case we do not require any specific interpretation. This is achieved by choosing

$$\begin{array}{ll}
\mathcal{T}_0 & := \{\mathrm{int}\} \\
\mathcal{D}_0 & := \{-2^{31}, \ldots, 2^{31} - 1\} \\
\delta_0(x) & := \mathrm{int} \quad \text{for all } x \in \mathcal{D}_0 \\
D_0(\mathit{plus}) & := \mathcal{D}_0 \times \mathcal{D}_0 \\
D_0(\mathit{div}) & := \mathcal{D}_0 \times (\mathcal{D}_0 \setminus \{0\}) \\
\mathcal{I}_0(\mathit{plus})(x, y) & := x + y \quad \text{(with Java overflow)} \\
\mathcal{I}_0(\mathit{div})(x, y) & := x/y \quad \text{(with Java overflow and rounding)}
\end{array}$$

We have not yet defined exactly what it means for some model to adhere to the restrictions expressed by a partial model. In order to do this, we first define a refinement relation between partial models. Essentially, one partial model refines another if its restrictions are stronger, i.e., if it contains all the restrictions of the other, and possibly more. In particular, more functions and predicates may be fixed, as well as more types and larger parts of the domain. It is also possible to fix previously underspecified parts of the interpretation. However, any types, interpretations, etc. that were previously fixed must remain the same. This is captured by the following definition:

Definition 1.37. *A partial model* $(\mathcal{T}_1, \mathcal{D}_1, \delta_1, D_1, \mathcal{I}_1)$ refines *another partial model* $(\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$*, if*

- $\mathcal{T}_1 \supseteq \mathcal{T}_0$,
- $\mathcal{D}_1 \supseteq \mathcal{D}_0$,

- $\delta_1(d) = \delta_0(d)$ *for all* $d \in \mathcal{D}_0$,
- $D_1(f) \supseteq D_0(f)$ *for all* $f \in \mathrm{FSym}$,
- $D_1(p) \supseteq D_0(p)$ *for all* $p \in \mathrm{PSym}$,
- $\mathcal{I}_1(f)(d_1, \ldots, d_n) = \mathcal{I}_0(f)(d_1, \ldots, d_n)$ *for all* $(d_1, \ldots, d_n) \in D_0(f)$ *and* $f \in \mathrm{FSym}$, *and*
- $\mathcal{I}_1(p) \cap D_0(p) = \mathcal{I}_0(p)$ *for all* $p \in \mathrm{PSym}$.

Example 1.38. We define a partial model that refines the one in the previous example by also fixing the interpretation of *zero*, and by restricting the division of zero by zero to give one.

$$
\begin{array}{ll}
\mathcal{T}_1 & := \{\mathrm{int}\} \\
\mathcal{D}_1 & := \{-2^{31}, \ldots, 2^{31} - 1\} \\
\delta_1(x) & := \mathrm{int} \quad \text{for all } x \in \mathcal{D}_0 \\
D_1(\mathit{zero}) & := \{()\} \quad \text{(the empty tuple)} \\
D_1(\mathit{plus}) & := \mathcal{D}_0 \times \mathcal{D}_0 \\
D_1(\mathit{div}) & := (\mathcal{D}_0 \times (\mathcal{D}_0 \setminus \{0\})) \cup \{(0,0)\} \\
\mathcal{I}_1(\mathit{zero})() & := 0 \\
\mathcal{I}_1(\mathit{plus})(x,y) & := x + y \quad \text{(with JAVA overflow)} \\
\mathcal{I}_1(\mathit{div})(x,y) & := \begin{cases} 1 & \text{if } x = y = 0, \\ x/y & \text{otherwise (with JAVA overflow and rounding)} \end{cases}
\end{array}
$$

To relate models to partial models, we can simply see models as a special kind of partial model in which all interpretations are completely fixed:

Definition 1.39. *Let* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *be a type hierarchy. Any* model $(\mathcal{D}, \delta, \mathcal{I})$ *may also be regarded as a* partial model $(\mathcal{T}_d, \mathcal{D}, \delta, D, \mathcal{I})$, *by letting* $D(f) = \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$ *for all function symbols* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$, *and* $D(p) = \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$ *for all predicate symbols* $p : A_1, \ldots, A_n \in \mathrm{PSym}$.

The models are special among the partial models in that they cannot be refined any further.

It is now clear how to identify models which adhere to the restrictions expressed in some partial model: we want exactly those models which are refinements of that partial model. To express that we are only interested in such models, we can *relativise* our definitions of validity, etc.

Definition 1.40. *Let a fixed type hierarchy and signature be given. Let* $\mathcal{M}_0$ *be a partial model.*

- *A formula* ϕ *is* logically valid with respect to $\mathcal{M}_0$ *if* $\mathcal{M}, \beta \models \phi$ *for any model* $\mathcal{M}$ *that refines* $\mathcal{M}_0$ *and any variable assignment* β.
- *A formula* ϕ *is* satisfiable with respect to $\mathcal{M}_0$ *if* $\mathcal{M}, \beta \models \phi$ *for some model* $\mathcal{M}$ *that refines* $\mathcal{M}_0$ *and some variable assignment* β.
- *A formula is* unsatisfiable with respect to $\mathcal{M}_0$ *if it is not satisfiable with respect to* $\mathcal{M}_0$.

Example 1.41. Even though division is often thought of as a partial function, which is undefined for the divisor 0, from the standpoint of our logic, a division by zero certainly produces a value. So the formula

$$\forall x.\exists y.\mathit{div}(x, \mathit{zero}) \doteq y$$

is logically valid, simply because for any value of x, one can interpret the term $\mathit{div}(x, \mathit{zero})$ and use the result as instantiation for y.

If we add constants *zero*, *one*, *two*, etc. with the obvious interpretations to the partial model of Example 1.36, then formulae like

$$\mathit{plus}(\mathit{one}, \mathit{two}) \doteq \mathit{three}$$

and

$$\mathit{div}(\mathit{four}, \mathit{two}) \doteq \mathit{two}$$

are logically valid with respect to that partial model, though they are not logically valid in the sense of Def. 1.28. However, it is not possible to add another fixed constant c to the partial model, such that

$$\mathit{div}(\mathit{one}, \mathit{zero}) \doteq c$$

becomes logically valid w.r.t. the partial model, since it does not fix the interpretation of the term $\mathit{div}(\mathit{one}, \mathit{zero})$. Therefore, for any given fixed interpretation of the constant c there is a model $(\mathcal{D}, \delta, \mathcal{I})$ that refines the partial model and that interprets $\mathit{div}(\mathit{one}, \mathit{zero})$ to something different, i.e.,

$$\mathcal{I}(\mathit{div})(1, 0) \neq \mathcal{I}(c)$$

So instead of treating *div* as a partial function, it is left *under-specified* in the partial model. Note that we handled the interpretation of "undefined" type casts in exactly the same way. See the sidebar on handling undefinedness (p. **??**) for a discussion of this approach to partiality.

For the next two sections, we will not be talking about partial models or relative validity, but only about logical validity in the normal sense. We will however come back to partial models in Section 1.7.

1.5 A Calculus

We have seen in the examples after Definition 1.28 how a formula can be shown to be logically valid, using mathematical reasoning about models, the definitions of the semantics, etc. The proofs given in these examples are however somewhat unsatisfactory in that they do not seem to be constructed in any systematic way. Some of the reasoning seems to require human intuition and resourcefulness. In order to use logic on a computer, we need a more

systematic, algorithmic way of discovering whether some formula is valid. A direct application of our semantic definitions is not possible, since for infinite universes, in general, an infinite number of cases would have to be checked.

For this reason, we now present a *calculus* for our logic. A calculus describes a certain arsenal of purely syntactic operations to be carried out on formulae, allowing us to determine whether a formula is valid. More precisely, to rule out misunderstandings from the beginning, *if* a formula is valid, we are able to establish its validity by systematic application of our calculus. If it is invalid, it might be impossible to detect this using this calculus. Also note that the calculus only deals with logical validity in the sense of Def. 1.28, and not with validity w.r.t. some partial model. We will come back to these questions in Section 1.6 and 1.7.

The calculus consists of "rules" (see Fig. 1.2, 1.3, and 1.4), along with some definitions that say how these rules are to be applied to decide whether a formula is logically valid. We now present these definitions and explain most of the rules, giving examples to illustrate their use.

The basic building block to which the rules of our calculus are applied is the *sequent*, which is defined as follows:

Definition 1.42. *A* sequent *is a pair of sets of closed formulae written as*

$$\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n \ .$$

The formulae ϕ_i on the left of the sequent arrow $\Longrightarrow$ *are called the* antecedent*, the formulae ψ_j on the right the* succedent *of the sequent. We use capital Greek letters to denote several formulae in the antecedent or succedent of a sequent, so by*

$$\Gamma, \phi \Longrightarrow \psi, \Delta$$

we mean a sequent containing ϕ in the antecedent, and ψ in the succedent, as well as possibly many other formulae contained in Γ, and Δ.

Note 1.43. Some authors define sequents using lists (sequences) or multi-sets of formulae in the antecedent or succedent. For us, sets are sufficient. So the sequent $\phi \Longrightarrow \phi, \psi$ is the same as $\phi, \phi \Longrightarrow \psi, \phi$.

Note 1.44. We do not allow formulae with free variables in our sequents. Free variables add technical difficulties and notoriously lead to confusion, since they have historically been used for several different purposes. Our formulation circumvents these difficulties by avoiding free variables altogether and sticking to closed formulae.

The intuitive meaning of a sequent

$$\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n$$

is the following:

> Whenever *all* the ϕ_i of the antecedent are true, then *at least one* of the ψ_j of the succedent is true.

Equivalently, we can read it as:

> It cannot be that *all* the ϕ_i of the antecedent are true, and *all* ψ_j of the succedent are false.

This whole statement represented by the sequent has to be shown *for all models*. If it can be shown for some model, we also say that the sequent is valid in that model. Since all formulae are closed, variable assignments are not important here. If we are simply interested in the logical validity of a single formula ϕ, we start with the simple sequent

$$\Longrightarrow \phi$$

and try to construct a proof. Before giving the formal definition of what exactly constitutes a proof, we now go through a simple example.

1.5.1 An Example Proof

We proceed by applying the rules of the calculus to construct a *tree of sequents*. We demonstrate this by a proof of the validity of the formula

$$(p \;\&\; q) \mathrel{->} (q \;\&\; p)$$

where p and q are predicates with no arguments.[13] We start with

$$\Longrightarrow (p \;\&\; q) \mathrel{->} (q \;\&\; p) \;\; .$$

In Fig. 1.2, we see a rule

$$\mathsf{impRight}\;\frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \mathrel{->} \psi, \Delta} \;\; .$$

impRight is the name of the rule. It serves to handle implications in the succedent of a sequent. The sequent below the line is the *conclusion* of the rule, and the one above is its *premiss*. Some rules in Fig. 1.2 have several or no premisses, we will come to them later.

The meaning of the rule is that if a sequent of the form of the premiss is valid, then the conclusion is also valid. We use it in the opposite direction: to prove the validity of the conclusion, it suffices to prove the premiss. We now apply this rule to our sequent, and write the result as follows:

$$\frac{(p \;\&\; q) \Longrightarrow (q \;\&\; p)}{\Longrightarrow (p \;\&\; q) \mathrel{->} (q \;\&\; p)}$$

[13] Such predicates are sometimes called *propositional variables*, but they should not be confused with the variables of our logic.

In this case, we take $p \;\&\; q$ for the ϕ in the rule and $q \;\&\; p$ for the ψ, with both Γ and Δ being empty.[14] There are now two rules in Fig. 1.2 that may be applied, namely andLeft and andRight. Let us use andLeft first. We add the result to the top of the previous proof:

$$\dfrac{\dfrac{p, q \Longrightarrow q \;\&\; p}{p \;\&\; q \Longrightarrow q \;\&\; p}}{\Longrightarrow (p \;\&\; q) \mathrel{->} (q \;\&\; p)}$$

In this case Γ contains the untouched formula $q \;\&\; p$ of the succedent. Now, we apply andRight. Since this rule has two premisses, our proof *branches*.

$$\dfrac{\dfrac{\dfrac{p, q \Longrightarrow q \qquad p, q \Longrightarrow p}{p, q \Longrightarrow q \;\&\; p}}{p \;\&\; q \Longrightarrow q \;\&\; p}}{\Longrightarrow (p \;\&\; q) \mathrel{->} (q \;\&\; p)}$$

A rule with several premisses means that its conclusion is valid if all of the premisses are valid. We thus have to show the validity of the two sequents above the topmost line. We can now use the close rule on both of these sequents, since each has a formula occurring on both sides.

$$\dfrac{\dfrac{\dfrac{\overline{p, q \Longrightarrow q} \qquad \overline{p, q \Longrightarrow p}}{p, q \Longrightarrow q \;\&\; p}}{p \;\&\; q \Longrightarrow q \;\&\; p}}{\Longrightarrow (p \;\&\; q) \mathrel{->} (q \;\&\; p)}$$

The close rule has no premisses, which means that the goal of a branch where it is applied is successfully proven. We say that the branch is *closed*. We have applied the close rule on all branches, so that was it! All branches are closed, and therefore the original formula was logically valid.

1.5.2 Ground Substitutions

Before discussing the workings of our calculus in a more rigorous way, we introduce a construct known as *substitution*. Substitutions are used by many of the rules that have to do with quantifiers, equality, etc.

Definition 1.45. *A* ground substitution *is a function τ that assigns a ground term to some finite set of variable symbols* $\mathrm{dom}(\tau) \subseteq \mathrm{VSym}$*, the* domain *of the substitution, with the restriction that*

[14] Γ, Δ, ϕ, ψ in the rule are place holders, also known as schema variables. The act of assigning concrete terms, formulae, or formula sets to schema variables is known as *matching*. See also Note 1.51 and Chapter **??** for details about pattern matching.

if $v \in \mathrm{dom}(\tau)$ *for a variable* $v{:}B \in \mathrm{VSym}$*, then* $\tau(v) \in \mathrm{Trm}_A$*, for some* A *with* $A \sqsubseteq B$*.*

We write $\tau = [u_1/t_1, \ldots, u_n/t_n]$ *to denote the particular substitution defined by* $\mathrm{dom}(\tau) = \{u_1, \ldots, u_n\}$ *and* $\tau(u_i) := t_i$*.*

We denote by τ_x *the result of removing a variable from the domain of* τ*, i.e.,* $\mathrm{dom}(\tau_x) := \mathrm{dom}(\tau) \setminus \{x\}$ *and* $\tau_x(v) := \tau(v)$ *for all* $v \in \mathrm{dom}(\tau_x)$*.*

Example 1.46. Given the signature from the previous examples,

$$\tau = [o/\mathit{empty}, n/\mathit{length}(\mathit{empty})]$$

is a substitution with

$$\mathrm{dom}(\tau) = \{o, n\} \ .$$

Note that the static type of *empty* is List, which is a subtype of Object, which is the type of the variable o. For this substitution, we have

$$\tau_o = [n/\mathit{length}(\mathit{empty})]$$

and

$$\tau_n = [o/\mathit{empty}] \ .$$

We can also remove both variables from the domain of τ, which gives us

$$(\tau_o)_n = [] \ ,$$

the empty substitution with $\mathrm{dom}([]) = \emptyset$. Removing a variable that is not in the domain does not change τ:

$$\tau_a = \tau = [o/\mathit{empty}, n/\mathit{length}(\mathit{empty})] \ .$$

The following is *not* a substitution:

$$[n/\mathit{empty}] \ ,$$

since the type List of the term *empty* is not a subtype of int, which is the type of the variable n.

Note 1.47. In Section **??**, a more general concept of substitution is introduced, that also allows substituting terms with free variables. This can lead to various complications that we do not need to go into at this point.

We want to apply substitutions not only to variables, but also to terms and formulae.

Definition 1.48. *The application of a ground substitution* τ *is extended to non-variable terms by the following definitions:*

- $\tau(x) := x$ *for a variable* $x \notin \mathrm{dom}(\tau)$*.*

- $\tau(f(t_1, \ldots, t_n)) := f(\tau(t_1), \ldots, \tau(t_n))$.

The application of a ground substitution τ to a formula is defined by

- $\tau(p(t_1, \ldots, t_n)) := p(\tau(t_1), \ldots, \tau(t_n))$.
- $\tau(\text{true}) := \text{true}$ *and* $\tau(\text{false}) := \text{false}$.
- $\tau(!\,\phi) := !(\tau(\phi))$,
- $\tau(\phi \;\&\; \psi) := \tau(\phi) \;\&\; \tau(\psi)$, *and correspondingly for* $\phi \mid \psi$ *and* $\phi \mathrel{->} \psi$.
- $\tau(\forall x.\phi) := \forall x.\tau_x(\phi)$ *and* $\tau(\exists x.\phi) := \exists x.\tau_x(\phi)$.

Example 1.49. Let's apply the ground substitution

$$\tau = [o/\mathit{empty}, n/\mathit{length}(\mathit{empty})]$$

from the previous example to some terms and formulae:

$$\begin{aligned} \tau(\mathit{plus}(n, n)) &= \mathit{plus}(\mathit{length}(\mathit{empty}), \mathit{length}(\mathit{empty})) \ , \\ \tau(n \doteq \mathit{length}((\text{List})o)) &= (\mathit{length}(\mathit{empty}) \doteq \mathit{length}((\text{List})\mathit{empty})) \ . \end{aligned}$$

By the way, this is an example of why we chose to use the symbol $\doteq$ instead of $=$ for the equality symbol in our logic. Here is an example with a quantifier:

$$\tau(\exists o.n \doteq \mathit{length}((\text{List})o)) = (\exists o.(\mathit{length}(\mathit{empty}) \doteq \mathit{length}((\text{List})o)) \ .$$

We see that the quantifier for o prevents the substitution from acting on the o inside its scope.

Some of our rules call for formulae *of the form* $[z/t](\phi)$ for some formula ϕ, variable z, and term t. In these cases, the rule is applicable to any formula that can be written in this way. Consider for instance the following rule from Fig. 1.3:

$$\mathsf{eqLeft}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta} \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

Looking at the conclusion, it requires two formulae

$$t_1 \doteq t_2 \quad \text{and} \quad [z/t_1](\phi)$$

in the antecedent. The rule adds the formula

$$[z/t_2](\phi)$$

to the antecedent of the sequent. Now consider the formulae

$$\mathit{length}(\mathit{empty}) \doteq \mathit{zero} \quad \text{and} \quad \mathit{length}(\mathit{empty}) \sqsubseteq \text{int} \ .$$

The right formula can also be written as

$$\mathit{length}(\mathit{empty}) \sqsubseteq \text{int} \ = \ [z/\mathit{length}(\mathit{empty})](z \sqsubseteq \text{int}) \ .$$

In other words, in this example, we have:

$$t_1 = \mathit{length}(\mathit{empty})$$
$$t_2 = \mathit{zero}$$
$$\phi = z \sqsubseteq \mathrm{int}$$

Essentially, the z in ϕ marks an occurrence of t_1 in the formula $[z/t_1](\phi)$. The new formula added by the rule, $[z/t_2](\phi)$, is the result of replacing this occurrence by the term t_2.

We do not exclude the case that there are no occurrences of the variable z in ϕ, or that there are several occurrences. In the case of no occurrences, $[z/t_1](\phi)$ and $[z/t_2](\phi)$ are the same formula, so the rule application does not do anything. In the case of several occurrences, we replace several instances of t_1 by t_2 simultaneously.

Note that this is just an elegant, yet precise way of formulating our calculus rules. In the implementation of the KeY system, it is more convenient to replace one occurrence at a time.

1.5.3 Sequent Proofs

As we saw in the example of Section 1.5.1, a sequent proof is a tree that is constructed according to a certain set of rules. This is made precise by the following definition:

Definition 1.50. *A* proof tree *is a finite tree (shown with the root at the bottom), such that*

- *each node of the tree is annotated with a sequent*
- *each inner node of the tree is additionally annotated with one of those rules shown in Figs. 1.2, 1.3, and 1.4 that have at least one premiss. This rule relates the node's sequent to the sequents of its descendants. In particular, the number of descendants is the same as the number of premisses of the rule.*
- *a leaf node may or may not be annotated with a rule. If it is, it is one of the rules that have no premisses, also known as closing rules.*

A proof tree for a formula ϕ *is a proof tree where the root sequent is annotated with* $\Longrightarrow \phi$.

A branch *of a proof tree is a path from the root to one of the leaves. A branch is* closed *if the leaf is annotated with one of the closing rules. A proof tree is* closed *if all its branches are closed, i.e., every leaf is annotated with a closing rule.*

A closed proof tree (for a formula ϕ) is also called a proof *(for ϕ).*

Note 1.51. A really rigorous definition of the concept of a proof would require a description of the *pattern matching* and replacement process that underlies

the application of the rules. This is done to a certain extent in Chapter **??**. For the time being, we assume that the reader understands that the Latin and Greek letters Γ, t_1, ϕ, z, A are actually place holders for arbitrary terms, formulae, types, etc. according to their context.

In a sense, models and proofs are complementary: to show that a formula is satisfiable, one has to describe a single model that satisfies it, as we did for instance in Example 1.32. To show that a formula is logically valid, we have previously shown that it is valid in any model, like for instance in Example 1.33. Now we can show logical validity by constructing a single proof.

1.5.4 The Classical First-Order Rules

Two rules in Fig. 1.2 carry a strange requirement: allRight and exRight require the choice of "$c : \rightarrow A$ a new constant, if $x{:}A$". The word "new" in this requirement means that the symbol c has not occurred in any of the sequents of the proof tree built so far. The idea is that to prove a statement for all x, one chooses an arbitrary but fixed c and proves the statement for that c. The symbol needs to be new since we are not allowed to assume anything about c (except its type).

If we use the calculus in the presence of a partial model in the sense of Section 1.4.3, we may only take a symbol c that is not fixed, i.e., $D_0(c) = \emptyset$. The reason is again to make sure that no knowledge about c can be assumed.

In order to permit the construction of proofs of arbitrary size, it is sensible to start with a signature that contains enough constant symbols of every type. We call signatures where this is the case "admissible":

Definition 1.52. *For any given type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, an* admissible signature *is a signature that contains an infinite number of constant symbols* $c :\rightarrow A$ *for every non-empty type* $A \in \mathcal{T}_q$*.*

Since the validity or satisfiability of a formula cannot change if symbols are added to the signature, it never hurts to assume that our signature is admissible. And in an admissible signature, it is always possible to pick a new constant symbol of any type.

We start our demonstration of the rules with some simple first-order proofs. We assume the minimal type hierarchy that consists only of $\bot$ and $\top$, see Example 1.7.

Example 1.53. Let the signature contain a predicate $p : \top$ and two variables $x{:}\top, y{:}\top$. We also assume an infinite set of constants $c_1, c_2, \ldots : \top$. We construct a proof for the formula

$$\exists x.\forall y.(p(x) \text{ -> } p(y)) \ .$$

We start with the sequent

$$\mathsf{andLeft}\ \frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \,\&\, \psi \Longrightarrow \Delta} \qquad \mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \,\&\, \psi, \Delta}$$

$$\mathsf{orRight}\ \frac{\Gamma \Longrightarrow \phi, \psi, \Delta}{\Gamma \Longrightarrow \phi \mid \psi, \Delta} \qquad \mathsf{orLeft}\ \frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \mid \psi \Longrightarrow \Delta}$$

$$\mathsf{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \mathrel{->} \psi, \Delta} \qquad \mathsf{impLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \mathrel{->} \psi \Longrightarrow \Delta}$$

$$\mathsf{notLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta}{\Gamma, !\,\phi \Longrightarrow \Delta} \qquad \mathsf{notRight}\ \frac{\Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow !\,\phi, \Delta}$$

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow [x/c](\phi), \Delta}{\Gamma \Longrightarrow \forall x.\phi, \Delta}$$
with $c : \to A$ a new constant, if $x{:}A$

$$\mathsf{allLeft}\ \frac{\Gamma, \forall x.\phi, [x/t](\phi) \Longrightarrow \Delta}{\Gamma, \forall x.\phi \Longrightarrow \Delta}$$
with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{exLeft}\ \frac{\Gamma, [x/c](\phi) \Longrightarrow \Delta}{\Gamma, \exists x.\phi \Longrightarrow \Delta}$$
with $c : \to A$ a new constant, if $x{:}A$

$$\mathsf{exRight}\ \frac{\Gamma \Longrightarrow \exists x.\phi, [x/t](\phi), \Delta}{\Gamma \Longrightarrow \exists x.\phi, \Delta}$$
with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{close}\ \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

$$\mathsf{closeFalse}\ \frac{}{\Gamma, \mathrm{false} \Longrightarrow \Delta} \qquad \mathsf{closeTrue}\ \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

Fig. 1.2. Classical first-order rules

$$\Longrightarrow \exists x.\forall y.(p(x) \mathrel{->} p(y))$$

for which only the exRight rule is applicable. We need to choose a term t for the instantiation. For lack of a better candidate, we take c_1:[15]

$$\frac{\Longrightarrow \forall y.(p(c_1) \mathrel{->} p(y)), \exists x.\forall y.(p(x) \mathrel{->} p(y))}{\Longrightarrow \exists x.\forall y.(p(x) \mathrel{->} p(y))}$$

Note that the original formula is left in the succedent. This means that we are free to choose a more suitable instantiation later on. For the time being, we apply the allRight rule, picking c_2 as the new constant.

[15] There are two reasons for insisting on admissible signatures: one is to have a sufficient supply of new constants for the allRight and exLeft rules. The other is that exRight and allLeft sometimes need to be applied although there is no suitable ground term in the sequent itself, as is the case here.

$$\dfrac{\dfrac{\dfrac{\Longrightarrow p(c_1) \dashrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}{\Longrightarrow \forall y.(p(c_1) \dashrightarrow p(y)), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) \dashrightarrow p(y))}}$$

Next, we apply impRight:

$$\dfrac{\dfrac{\dfrac{\dfrac{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}{\Longrightarrow p(c_1) \dashrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \forall y.(p(c_1) \dashrightarrow p(y)), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) \dashrightarrow p(y))}}$$

Since the closing rule close cannot be applied to the leaf sequent (nor any of the other closing rules), our only choice is to apply exRight again. This time, we choose the term c_2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(c_1) \Longrightarrow \forall y.(p(c_2) \dashrightarrow p(y)), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow p(c_1) \dashrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \forall y.(p(c_1) \dashrightarrow p(y)), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) \dashrightarrow p(y))}}$$

Another application of allRight (with the new constant c_3) and then impRight give us:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(c_1), p(c_2) \Longrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}{p(c_1) \Longrightarrow p(c_2) \dashrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{p(c_1) \Longrightarrow \forall y.(p(c_2) \dashrightarrow p(y)), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow p(c_1) \dashrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \forall y.(p(c_1) \dashrightarrow p(y)), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) \dashrightarrow p(y))}}$$

Finally, we see that the atom $p(c_2)$ appears on both sides of the sequent, so we can apply the close rule

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p(c_1), p(c_2) \Longrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{p(c_1) \Longrightarrow p(c_2) \dashrightarrow p(c_3), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{p(c_1) \Longrightarrow \forall y.(p(c_2) \dashrightarrow p(y)), p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{p(c_1) \Longrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow p(c_1) \dashrightarrow p(c_2), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \forall y.(p(c_1) \dashrightarrow p(y)), \exists x.\forall y.(p(x) \dashrightarrow p(y))}}{\Longrightarrow \exists x.\forall y.(p(x) \dashrightarrow p(y))}}$$

This proof tree has only one branch, and a closing rule has been applied to the leaf of this branch. Therefore, all branches are closed, and this is a proof for the formula $\exists x.\forall y.(p(x) \mathrel{->} p(y))$.

Example 1.54. We now show an example of a branching proof. In order to save space, we mostly just write the leaf sequents of the branch we are working on.

We take again the minimal type hierarchy. The signature contains two predicate symbols $p, q : \top$, as well as the infinite set of constants $c_1, c_2, \ldots : \top$ and a variable $x{:}\top$. We show the validity of the formula

$$(\exists x.p(x) \mathrel{->} \exists x.q(x)) \mathrel{->} \exists x.(p(x) \mathrel{->} q(x)) \ .$$

We start with the sequent

$$\Longrightarrow (\exists x.p(x) \mathrel{->} \exists x.q(x)) \mathrel{->} \exists x.(p(x) \mathrel{->} q(x))$$

from which the impRight rule makes

$$\exists x.p(x) \mathrel{->} \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{->} q(x)) \ .$$

We now apply impLeft, which splits the proof tree. The proof tree up to this point is:

$$\dfrac{\dfrac{\Longrightarrow \exists x.p(x), \exists x.(p(x) \mathrel{->} q(x)) \qquad \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{->} q(x))}{\exists x.p(x) \mathrel{->} \exists x.q(x) \Longrightarrow \exists x.(p(x) \mathrel{->} q(x))}}{\Longrightarrow (\exists x.p(x) \mathrel{->} \exists x.q(x)) \mathrel{->} \exists x.(p(x) \mathrel{->} q(x))}$$

On the left branch, we have to choose a term to instantiate one of the existential quantifiers. It turns out that any term will do the trick, so we apply exRight with c_1 on $\exists x.p(x)$, to get

$$\Longrightarrow p(c_1), \exists x.p(x), \exists x.(p(x) \mathrel{->} q(x))$$

and then on $\exists x.(p(x) \mathrel{->} q(x))$, which gives

$$\Longrightarrow p(c_1), p(c_1) \mathrel{->} q(c_1), \exists x.p(x), \exists x.(p(x) \mathrel{->} q(x)) \ .$$

We now apply impRight to get

$$p(c_1) \Longrightarrow p(c_1), q(c_1), \exists x.p(x), \exists x.(p(x) \mathrel{->} q(x))$$

to which the close rule applies.

On the right branch, we apply exLeft using c_2 as the new constant, which gives us

$$q(c_2) \Longrightarrow \exists x.(p(x) \mathrel{->} q(x)) \ .$$

We now use exRight with the instantiation c_2, giving

$$q(c_2) \Longrightarrow p(c_2) \mathrel{->} q(c_2), \exists x.(p(x) \mathrel{->} q(x)) \ .$$

impRight now produces

$$q(c_2), p(c_2) \Longrightarrow q(c_2), \exists x.(p(x) \mathrel{->} q(x)) \ ,$$

to which close may be applied.

To summarise, for each of !, &, |, –>, ∀, and ∃, there is one rule to handle occurrences in the antecedent and one rule for the succedent. The only "indeterminisms" in the calculus are 1. the order in which the rules are applied, and 2. the instantiations chosen for allRight and exRight.

Both of these indeterminisms are of the kind known as *don't care indeterminism*. What this means is that any choice of rule application order or instantiations can at worst delay (maybe infinitely) the closure of a proof tree. If there is a closed proof tree for a formula, any proof tree can be completed to a closed proof tree. It is not necessary in principle to backtrack over rule applications, there are no "dead ends" in the search space. A calculus with this property is known as *proof confluent*.

It should be noted that an unfortunate choice of applied rules can make the resulting proof much larger in practice, so that it can be worthwhile to remove part of a proof attempt and to start from the beginning.

1.5.5 The Equality Rules

The essence of reasoning about equality is the idea that if one entity equals another, then any occurrence of the first may be replaced by the second. This idea would be expressed by the following (in general incorrect) rule:

$$\mathsf{eqLeftWrong}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$

Unfortunately in the presence of subtyping, things are not quite that easy. Assume for instance a type hierarchy with two types $B \sqsubseteq A$, but $B \neq A$, and a signature containing constants $a :\rightarrow A$ and $b :\rightarrow B$, and a predicate $p : B$. If we apply the above rule on the sequent

$$b \doteq a, p(b) \Longrightarrow$$

we get the new "sequent"

$$b \doteq a, p(b), p(a) \Longrightarrow \quad .$$

This is in fact not a sequent, since $p(a)$ is not a formula, because p cannot be applied to a term of static type A.

There are two ways to solve this problem. The first way is embodied by the rules eqLeft and eqRight in Fig. 1.3: The static type of the new term t_2 is required to be a subtype of the type of the original t_1. This guarantees that the resulting formula is still well-typed. Indeed, it would have forbidden the erroneous rule application of our example since $\sigma(t_2) \not\sqsubseteq \sigma(t_1)$.

The other solution is to insert a cast. If $t_1 \doteq t_2$ holds, and A is the static type of t_1, then $t_2 \doteq (A)t_2$ also holds, and therefore $t_1 \doteq (A)t_2$, so we can

$$\mathsf{eqLeft}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta} \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

$$\mathsf{eqRight}\ \frac{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_1](\phi), \Delta} \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1)$$

$$\mathsf{eqLeft'}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/(A)t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta} \quad \text{with } A := \sigma(t_1)$$

$$\mathsf{eqRight'}\ \frac{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/(A)t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_1](\phi), \Delta} \quad \text{with } A := \sigma(t_1)$$

$$\mathsf{eqSymmLeft}\ \frac{\Gamma, t_2 \doteq t_1 \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow \Delta} \qquad \mathsf{eqClose}\ \frac{}{\Gamma \Longrightarrow t \doteq t, \Delta}$$

Fig. 1.3. Equality rules

rewrite t_1 to $(A)t_2$, which still has the static type A, so again, the formula remains well-typed. This is what the rules eqLeft′ and eqRight′ do.[16]

Note that the equation $t_1 \doteq t_2$ has to be on the left of the sequent for all four of these rules. The difference between the Left and Right versions is the position of the formula on which the equation is applied. The only way of handling an equation in the succedent, i.e., of *showing* that an equation holds is to apply other equations on both sides until they become identical, and then to apply eqClose.

In general, one might want to apply equations in both directions, i.e., also rewrite t_2 to t_1. We allow this by the rule eqSymmLeft. Equivalently we could have given variants of the four rewriting rules, that apply equations from right to left, but that would have doubled the number of rules.

Example 1.55. Assume a type hierarchy containing two types $B \sqsubseteq A$ in addition to $\bot$ and $\top$. We need two constants $a : \rightarrow A$ and $b : \rightarrow B$, functions $f : B \rightarrow B$ and $g : A \rightarrow B$, and a variable $x{:}A$. We show the validity of

$$(\forall x.f(g(x)) \doteq g(x)\ \&\ b \doteq g(a)) \mathrel{-\!\!>} f(f(b)) \doteq f(g(a))\ .$$

[16] As is illustrated in Example 1.58, any application of these two rules may be replaced by a series of applications of other rules, so it would be possible to do without them. Still, it is sometimes convenient to have them, since they allow to do all equality reasoning first, possibly inserting casts, and taking care of the type reasoning later.

Starting from the initial sequent

$$\Longrightarrow (\forall x.f(g(x)) \doteq g(x) \;\&\; b \doteq g(a)) \mathrel{->} f(f(b)) \doteq f(g(a)) \ ,$$

applying impRight and andLeft leads to

$$\forall x.f(g(x)) \doteq g(x), b \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

We now apply allLeft for the instantiation a. Since we do not need any more instances of the $\forall$ formula, we abbreviate it by "..." in the rest of this example:

$$\ldots, f(g(a)) \doteq g(a), b \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

Consider the equation $b \doteq g(a)$. The static type of both sides is B, so we could apply the equation in both directions. We would like to rewrite occurrences of $g(a)$ to the smaller term b, so we apply eqSymmLeft to turn the equation around:

$$\ldots, f(g(a)) \doteq g(a), g(a) \doteq b \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

Now we apply $g(a) \doteq b$ on the left side of the equation $f(g(a)) \doteq g(a)$. As we explained at the end of Section 1.5.2, this is done by marking the place where the equality should be applied by a variable z and "pulling out" the term t_1 into a substitution, i.e., $(f(z) \doteq g(a))[z/t_1]$. In other words, we apply eqLeft with

$$t_1 = g(a) \qquad t_2 = b \qquad \phi = f(z) \doteq g(a)$$

to get

$$\ldots, f(g(a)) \doteq g(a), g(a) \doteq b, f(b) \doteq g(a) \Longrightarrow f(f(b)) \doteq f(g(a)) \ .$$

The next step is to apply the new equation $f(b) \doteq g(a)$ on the occurrence of $f(b)$ in the succedent, i.e., we apply eqRight with

$$t_1 = f(b) \qquad t_2 = g(a) \qquad \phi = f(z) \doteq f(g(a))$$

to get

$$\begin{aligned} \ldots, f(g(a)) \doteq g(a), g(a) \doteq b, f(b) \doteq g(a) \\ \Longrightarrow f(g(a)) \doteq f(g(a)), f(f(b)) \doteq f(g(a)) \end{aligned}$$

which can be closed using the eqClose rule.

The eqLeft′/eqRight′ rules introduce casts which can only be treated by using some additional rules. We therefore postpone an example of their use to the following section.

The equality rules allow much more freedom in their application than the previously shown rules in Fig. 1.2. As a general guideline, it is often best to apply equations in the direction that makes terms smaller or simpler, provided this is allowed by the types.

It should be mentioned at this point that the equality rules in the implementation of the KeY system are organised in a slightly different way. Instead

of letting the user decide between the rules eqLeft and eqLeft′, or between eqRight and eqRight′ for an occurrence in the succedent, the system checks whether $\sigma(t_2) \sqsubseteq \sigma(t_1)$. If this is the case, no cast is needed and eqLeft, resp. eqRight is applied, otherwise a cast is inserted, corresponding to an application of eqLeft′, resp. eqRight′. This combined behaviour is achieved by a rule named applyEq (see Fig. **??**).

1.5.6 The Typing Rules

$$\text{typeEq}\ \frac{\Gamma, t_1 \doteq t_2, t_2 \sqsubseteq \sigma(t_1), t_1 \sqsubseteq \sigma(t_2) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow \Delta} \qquad \text{typeGLB}\ \frac{\Gamma, t \sqsubseteq A, t \sqsubseteq B, t \sqsubseteq A \sqcap B \Longrightarrow \Delta}{\Gamma, t \sqsubseteq A, t \sqsubseteq B \Longrightarrow \Delta}$$

$$\text{typeStatic}\ \frac{\Gamma, t \sqsubseteq \sigma(t) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\text{typeAbstract}\ \frac{\Gamma, t \sqsubseteq A, t \sqsubseteq B_1 \mid \cdots \mid t \sqsubseteq B_k \Longrightarrow \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow \Delta}$$

with $A \in \mathcal{T}_a$ and $B_1, \ldots, B_k$ the direct subtypes of A

$$\text{castAddLeft}\ \frac{\Gamma, [z/t](\phi), t \sqsubseteq A, [z/(A)t](\phi) \Longrightarrow \Delta}{\Gamma, [z/t](\phi), t \sqsubseteq A \Longrightarrow \Delta}\ \text{where } A \sqsubseteq \sigma(t). \qquad \text{castAddRight}\ \frac{\Gamma, t \sqsubseteq A \Longrightarrow [z/(A)t](\phi), [z/t](\phi), \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow [z/t](\phi), \Delta}\ \text{where } A \sqsubseteq \sigma(t).$$

$$\text{castDelLeft}\ \frac{\Gamma, [z/t](\phi), [z/(A)t](\phi) \Longrightarrow \Delta}{\Gamma, [z/(A)t](\phi) \Longrightarrow \Delta}\ \text{where } \sigma(t) \sqsubseteq A. \qquad \text{castDelRight}\ \frac{\Gamma \Longrightarrow [z/t](\phi), [z/(A)t](\phi), \Delta}{\Gamma \Longrightarrow [z/(A)t](\phi), \Delta}\ \text{where } \sigma(t) \sqsubseteq A.$$

$$\text{castTypeLeft}\ \frac{\Gamma, (A)t \sqsubseteq B, t \sqsubseteq A, t \sqsubseteq B \Longrightarrow \Delta}{\Gamma, (A)t \sqsubseteq B, t \sqsubseteq A \Longrightarrow \Delta} \qquad \text{castTypeRight}\ \frac{\Gamma, t \sqsubseteq A \Longrightarrow t \sqsubseteq B, (A)t \sqsubseteq B, \Delta}{\Gamma, t \sqsubseteq A \Longrightarrow (A)t \sqsubseteq B, \Delta}$$

$$\text{closeSubtype}\ \frac{}{\Gamma, t \sqsubseteq A \Longrightarrow t \sqsubseteq B, \Delta}\ \text{with } A \sqsubseteq B \qquad \text{closeEmpty}\ \frac{}{\Gamma, t \sqsubseteq \bot \Longrightarrow \Delta}$$

Fig. 1.4. Typing rules

The remaining rules, shown in Fig. 1.4, all concern type casts and type predicates. In problems where all terms are of the same type, and no casts or type predicates occur, these rules are not needed.

Given two terms $t_1 \in \mathrm{Trm}_A$ and $t_2 \in \mathrm{Trm}_B$ of static types A and B, the first rule allows deriving $t_2 \sqsubseteq A$ and $t_1 \sqsubseteq B$. Why is this allowed? Given some model $\mathcal{M}$, and variable assignment β, if $\mathcal{M}, \beta \models t_1 \doteq t_2$, then $\mathrm{val}_{\mathcal{M},\beta}(t_1) = \mathrm{val}_{\mathcal{M},\beta}(t_2)$. Therefore, the dynamic types of the terms' values are also equal: $\delta(\mathrm{val}_{\mathcal{M},\beta}(t_1)) = \delta(\mathrm{val}_{\mathcal{M},\beta}(t_2))$. Now, the dynamic type of each term is a subtype of the static type of the term. Since the dynamic types are the same, we additionally know that the dynamic type of each term is a subtype of the static type of *the other* term. Hence, $\mathcal{M}, \beta \models t_2 \sqsubseteq A$ and $\mathcal{M}, \beta \models t_1 \sqsubseteq B$. In combination with the typeStatic and typeGLB rules, we can go on by deriving $t_1 \sqsubseteq A \sqcap B$ and $t_2 \sqsubseteq A \sqcap B$.

The typeAbstract rule handles type predicate literals for abstract types. The underlying reasoning is that if the dynamic type of a value cannot be equal to an abstract type, so if $t \sqsubseteq A$ holds for an abstract type A, then $t \sqsubseteq B$ holds for some subtype B of A. Since we require type hierarchies to be finite, we can form the disjunction $t \sqsubseteq B_1 \mid \cdots \mid t \sqsubseteq B_k$ for all direct subtypes B_i of A.[17] If one of the direct subtypes B_i is itself abstract, the rule can be applied again on $t \sqsubseteq B_i$.

The castAdd, castType, and castDel rules can be used to close proof trees that involve formulae with type casts. More specifically, we need to deal with the situation that a branch can almost be closed, using for instance close or eqClose, but the involved formulae or terms are not quite equal, they differ by some of the casts. In general, the sequent also contains type predicates that allow to decide whether the casts are "successful" or not.

The basic observation is that if $t \sqsubseteq A$ holds, then the cast $(A)t$ does not change the value of t, so $(A)t \doteq t$ also holds. It is tempting to introduce rules like the following, which allows to remove casts in such situations:

$$\mathsf{wrongCastDelLeft}\ \frac{\Gamma, [z/(A)t](\phi), t \sqsubseteq A, [z/t](\phi) \Longrightarrow \Delta}{\Gamma, [z/(A)t](\phi), t \sqsubseteq A \Longrightarrow \Delta}$$

Unfortunately, the new formula $[z/t](\phi)$, in which the cast was removed, is possibly no longer well-typed: In general, the static type of t is a supertype of that of $(A)t$. Our solution to this problem is to *add* casts to the involved terms or formulae until they become equal. This is the purpose of the castAdd rules.

There are also castDel rules to delete casts, but these are only available if the static type of a term is a subtype of the type being cast to. In that case, the cast is obviously redundant, and removing it preserves the well-typedness of terms.

The two castType rules can be considered valid special cases of our wrongCastDelLeft rule: If we know $t \sqsubseteq A$, then we may remove the cast in $(A)t \sqsubseteq B$ to obtain $t \sqsubseteq B$. There is no problem with the static types here, since the type predicate $\sqsubseteq B$ may be applied to terms of arbitrary type. These rules

[17] B is a direct subtype of A if A and B are distinct types, $B \sqsubseteq A$, and there is no type C that is distinct from A and B with $B \sqsubseteq C \sqsubseteq A$, Def. **??**.

are occasionally needed to derive the most specific type information possible about the term t.

We now illustrate these rules in some examples.

Example 1.56. We start by the formula from Example 1.33. In any type hierarchy that contains some type A, and a signature with a variable $x{:}\top$ and a constant $c{:}\top$, we show the validity of

$$\forall x.((A)x \doteq x \mathrel{->} x \sqsubseteq A) \ .$$

The initial sequent is

$$\Longrightarrow \forall x.((A)x \doteq x \mathrel{->} x \sqsubseteq A) \ ,$$

on which we apply the allRight and impRight to get

$$(A)c \doteq c \Longrightarrow c \sqsubseteq A \ .$$

The static type of c is $\top$, and the static type of $(A)c$ is A. We apply the typeEq rule to get

$$(A)c \doteq c, c \sqsubseteq A, (A)c \sqsubseteq \top \Longrightarrow c \sqsubseteq A \ .$$

Since $c \sqsubseteq A$ appears on both sides, this can be closed using the close rule.

Example 1.57. With the same type hierarchy and signature as the previous example, we now show the converse implication:

$$\forall x.(x \sqsubseteq A \mathrel{->} (A)x \doteq x) \ .$$

Again, we apply allRight and impRight on the initial sequent, to obtain

$$c \sqsubseteq A \Longrightarrow (A)c \doteq c \ .$$

We now apply castAddRight with

$$t = c \qquad \text{and} \qquad \phi = (A)c \doteq z$$

to obtain

$$c \sqsubseteq A \Longrightarrow (A)c \doteq (A)c, (A)c \doteq c \ ,$$

which can be closed using eqClose.

Example 1.58. Here is a more complicated example of type reasoning. We return to the type hierarchy from Example 1.6, p. 6. Remember that

$$\text{AbstractList} = \text{AbstractCollection} \sqcap \text{List} \ .$$

The following functions are used:

$$\begin{aligned} &\mathit{ord} : \text{AbstractCollection} \to \text{AbstractList} \\ &\mathit{rev} : \text{List} \to \text{List} \end{aligned}$$

Maybe *ord* takes a collection and puts it into some order, whereas *rev* reverses a list. We also use a constant a:AbstractCollection. The problem is to show the validity of

$$ord(a) \doteq a \mathrel{-\!\!>} rev(ord(a)) \doteq rev((\text{List})a) \ .$$

In this example, we silently omit some formulae from sequents, if they are not needed any more, to make it easier to follow the development. After applying impRight on the initial sequent, we get

$$ord(a) \doteq a \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ . \qquad (*)$$

Next, we would like to rewrite $ord(a)$ to a in the succedent. However, the static type of a is AbstractCollection, which is not a subtype of the static type of $ord(a)$, namely AbstractList. Therefore, we must use eqRight′, which introduces a cast and gives us:

$$ord(a) \doteq a \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})a) \ .$$

Our goal must now be to make the two casts in the succedent equal. To deduce more information about the type of a, we apply typeEq on the left to get

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})a)$$

(we omit the other, uninteresting formula $ord(a) \sqsubseteq \text{AbstractCollection}$). Now, how do we replace the cast to List by a cast to AbstractList? We use a combination of two rules: First, we apply castAddRight to insert a cast:

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{List})(\text{AbstractList})a) \ .$$

Since AbstractList $\sqsubseteq$ List, the outer cast has become redundant, so we use castDelRight to remove it:

$$a \sqsubseteq \text{AbstractList} \Longrightarrow rev((\text{AbstractList})a) \doteq rev((\text{AbstractList})a) \ .$$

This sequent can be closed using eqClose.

It turns out that applications of the eqRight′/eqLeft′ rules can always be replaced by sequences of applications of the other rules. They were only added because they are sometimes more convenient. We demonstrate this by showing an alternative way of proceeding from the sequent $(*)$ above. We first apply the typeEq rule, which gives us

$$ord(a) \doteq a, a \sqsubseteq \text{AbstractList} \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ .$$

We can then use castAddRight on the right side of the equation in the antecedent, yielding

$$ord(a) \doteq (\text{AbstractList})a, a \sqsubseteq \text{AbstractList} \Longrightarrow rev(ord(a)) \doteq rev((\text{List})a) \ .$$

Now, the static types are the same on both sides and we can use eqRight to obtain

$$a \sqsubseteq \text{AbstractList} \Longrightarrow \mathit{rev}((\text{AbstractList})a) \doteq \mathit{rev}((\text{List})a) \ .$$

From this sequent, we continue as before.

Example 1.59. Using the type hierarchy from Example 1.6 once again, a variable l : List and a constant c:List, we show validity of

$$\forall l.l \sqsubseteq \text{ArrayList} \ .$$

This is of course due to the fact that ArrayList is the top-most non-abstract subtype of List. Starting from

$$\Longrightarrow \forall l.l \sqsubseteq \text{ArrayList} \ ,$$

we apply the rule allRight to obtain

$$\Longrightarrow c \sqsubseteq \text{ArrayList} \ .$$

We can use the typeStatic rule for c to get

$$c \sqsubseteq \text{List} \Longrightarrow c \sqsubseteq \text{ArrayList} \ .$$

Now typeAbstract produces

$$c \sqsubseteq \text{AbstractList} \Longrightarrow c \sqsubseteq \text{ArrayList} \ ,$$

since AbstractList is the only direct subtype of the abstract type List. Since AbstractList is also abstract, we apply typeStatic again, to get

$$c \sqsubseteq \text{ArrayList} \Longrightarrow c \sqsubseteq \text{ArrayList} \ ,$$

which can be closed by close.

Example 1.60. In the type hierarchy from Example 1.6, using a variable i : int, and constants c:int and *null*:Null, we show

$$!\,\exists i.i \doteq \mathit{null} \ .$$

On the initial sequent

$$\Longrightarrow !\,\exists i.i \doteq \mathit{null} \ ,$$

we apply notRight to obtain

$$\exists i.i \doteq \mathit{null} \Longrightarrow \ ,$$

and exRight, which gives

$$c \doteq \mathit{null} \Longrightarrow \ .$$

Using typeStatic and typeEq for c produces

$$c \doteq \mathit{null}, c \sqsubseteq \mathrm{int}, c \sqsubseteq \mathrm{Null} \Longrightarrow \quad .$$

The intersection of int and Null is the empty type, so we can use typeGLB to derive

$$c \doteq \mathit{null}, c \sqsubseteq \bot \Longrightarrow \ ,$$

which can be closed using closeEmpty.

To summarise, the general idea of type reasoning is to start by identifying the interesting terms t. For these terms, one tries to derive the most specific type information, i.e., a type predicate literal $t \sqsubseteq A$ where the type A is as small as possible with respect to $\sqsubseteq$, by using typeStatic and typeEq, etc. Then, add a cast to the most specific known type in front of the interesting occurrences of t. On the other hand, delete redundant casts using the castDel rules. Sometimes, branches can be closed due to contradictory type information using closeSubtype and closeEmpty.

1.6 Soundness, Completeness

At first sight, the rules given in Section 1.5 might seem like a rather haphazard collection. But in fact, they enjoy two important properties. First, it is not possible to close a proof tree for a formula that is not logically valid. This is known as *soundness*. Second, if a formula is logically valid, then there is always a proof for it. This property is known as *completeness*. These two properties are so important that we state them as a theorem.

Theorem 1.61. *Let a fixed type hierarchy and an admissible signature be given. Then any formula ϕ is logically valid if and only if there is a sequent proof for ϕ constructed according to Def. 1.50.*

A proof of this result has been given by Giese [2005].

It is important to note that the theorem does not state soundness and completeness for our notion of validity with respect to partial models. This issue is discussed further in Section 1.7.

Soundness is much more important than completeness, in the sense that more harm is usually done if a wrong statement is considered correct, than if a valid statement cannot be shown. For instance, if a proof for the correctness of a piece of critical software is produced, and the software is used in the belief that it is correct, the consequences might be catastrophic.

On the other hand, not being able to prove the correctness of a correct piece of software with a given method might delay its deployment. Maybe the verification can be done by some other method. Maybe the formal proof is not considered to be crucial.

In practice, however, when a proof attempt fails, it is good to know that there can be only two reasons: either the statement to be shown is not valid, or one has not looked hard enough for a proof. The possibility that the statement is valid, but no proof exists, would make the whole situation more confusing.

Since we ultimately intend to use our logic and calculus on a computer, where a program should help us to find the proofs, let us consider some of the computational aspects of our calculus.

We already mentioned that the calculus is proof confluent: If a formula ϕ is valid, then any proof tree for ϕ can be completed to a closed proof. No rule application can lead into a "dead end". However, it is still possible to expand a proof tree for a valid formula indefinitely without finding a closed proof, just by regularly performing the "wrong" rule applications.

The good news is that it *is* possible to apply rules systematically in such a way that a closed proof is eventually found if it exists. This leads to a "computational" version of the previous version:

Theorem 1.62. *Let a fixed type hierarchy and an admissible signature be given. There is a program with the following property: if it is given a formula as input, it terminates stating the validity of the input formula if and only if that formula is logically valid.*

What if the formula is not valid? In general, the program will search indefinitely, and never give any output. It is possible to show that this must be so: It is a property of our logic that there can be no program that terminates on both valid and invalid formulae and correctly states whether the input is valid.

The technical way of describing this situation is to say that the validity of formulae in our logic is *undecidable*. This means that there is no program that terminates on all inputs and answers the question of validity.

More precisely, validity is *semi-decidable*, which means that there *is* a program that at least gives a positive answer for valid formulae. We equivalently say that the set of valid formulae is *recursively enumerable*, which means that it is possible to write a program that prints a list of all valid formulae.

For the practical use of a theorem proving program, this means that if the program runs for a very long time, there is no way of knowing whether the statement we are trying to prove is wrong, or whether we just have to wait for an even longer time.

There are logics (propositional logic and some modal logics) that have a better behaviour in this respect: there are theorem provers which terminate on all input and answer whether the input is a valid formula. However, these logics are a lot less expressive, and therefore not suited for detailed descriptions of complex software systems. Any logic that is expressive enough for that purpose has an undecidable validity problem.

The interested reader can find much more information on all aspects of the mechanisation of reasoning in the Handbook of Automated Reasoning edited by Robinson and Voronkov [2001].

1.7 Incompleteness

The soundness and completeness properties stated in the previous section do not apply to validity relative to some partial model. Indeed, it is hard to give general results about relative validity since any set of operations with fixed interpretation would require its own set of additional rules.

In this section, we discuss a particularly important case, namely the operations of addition and multiplication on the natural numbers. We do not include subtraction or division, since they can be expressed in terms of addition and multiplication.

Let us assume that the type hierarchy contains a sort N and the signature contains function symbols

$$\begin{aligned} \mathit{zero} &: \rightarrow N \\ \mathit{succ} &: N \rightarrow N \\ \mathit{plus} &: N, N \rightarrow N \\ \mathit{times} &: N, N \rightarrow N \end{aligned}$$

The partial model that interests us is defined as

$$\begin{array}{ll} \mathcal{T}_0 & := \{N\} \\ \mathcal{D}_0 & := \mathbb{N} = \{0, 1, 2, \ldots\} \\ \delta_0(x) & := N \quad \text{for all } x \in \mathcal{D}_0 \\ D_0(\mathit{zero}) & := \{()\} \\ D_0(\mathit{succ}) & := \mathbb{N} \\ D_0(\mathit{plus}) & := D_0(\mathit{times}) := \mathbb{N} \times \mathbb{N} \\ \mathcal{I}_0(\mathit{zero})() & := 0 \\ \mathcal{I}_0(\mathit{succ})(x) & := x + 1 \\ \mathcal{I}_0(\mathit{plus})(x, y) & := x + y \\ \mathcal{I}_0(\mathit{times})(x, y) & := xy \end{array}$$

We call this model "arithmetic". Note that our domain $\mathcal{D}_0$ now contains all the mathematical integers, and not only the JAVA integers as in previous examples. With this signature and partial model, individual natural numbers can be expressed as zero for 0, $\mathit{succ}(\mathit{zero})$ for 1, $\mathit{succ}(\mathit{succ}(\mathit{zero}))$ for 2, etc. The operations of addition and multiplication are sufficient to express many interesting mathematical concepts. For instance the following formula with free variable x expresses that x is either zero, or one, or a prime number:

$$\forall y.\forall z.(\mathit{times}(y, z) \doteq x \mathrel{-\!\!>} y \doteq x \mid z \doteq x) \ ,$$

and the following the fact that there are infinitely many prime numbers:[18]

$$\forall x.\exists u.\forall y.\forall z.(\mathit{times}(y, z) \doteq \mathit{plus}(x, u) \mathrel{-\!\!>} y \doteq \mathit{plus}(x, u) \mid z \doteq \mathit{plus}(x, u)) \ .$$

[18] It expresses that for any x, one can find a u, such that $x + u$ is prime, in other words there are primes of arbitrary magnitude.

Due to their expressivity, these basic operations on numbers are among the first things one might want to fix in a partial model. The bad news is that there can be no complete calculus for validity with respect to arithmetic.

Theorem 1.63. *There is no set of sequent rules suitable for mechanisation*[19] *such that a formula ϕ is valid w.r.t arithmetic if and only if there is a closed sequent proof for ϕ using these rules.*

Indeed, there is no program with the following property: if it is given a formula as input, it terminates stating the validity of the input formula if and only if that formula is logically valid w.r.t. arithmetic.

This is essentially the famous Incompleteness Theorem of Gödel [1931]. It means that if we want the expressive power of arithmetic, there are always some theorems that are true, but cannot be shown in our calculus. Another way of expressing this is that any sound calculus is necessarily incomplete. Therefore, one also calls a logic with this property incomplete.

In practice, the way of dealing with this problem is to add a number of rules to the calculus that capture the behaviour of *plus* and *times*, as well as one particular rule called the *induction rule* (see also Chapter **??**). For instance, we can add the rules in Fig. 1.5. Most of these rules just add simple properties of the involved operations to the sequent. The interesting rule is natInduct: It expresses that if you can show that a statement holds for *zero*, and that if it holds for some number n, it also holds for the next number $succ(n)$, then it must hold for all numbers.

These rules are still subject to the incompleteness theorem. But it turns out that using these rules, it is possible to prove almost any arithmetical statement that occurs *in practice*. Virtually any theorem about natural numbers that occurs in mathematics is ultimately proven using some variation of these few rules.[20]

It is interesting to note that many of the data structures appearing in computer programs, like for instance lists, strings, or trees have the same properties. In fact their behaviour can be encoded using numbers, and on the other hand, they can be used to simulate arithmetic. Therefore, for these data types the same observation holds, namely that validity relative to them makes the logic incomplete, but adding an appropriate induction rule (structural induction) allows proving almost all *practically interesting* statements. Induction is discussed in much greater detail in Chapter **??**.

Fortunately however, this is in a sense the *only* kind of incompleteness one has to confront: as explained in Section **??**, the calculus used in KeY to reason about programs is complete *relative to arithmetic*, meaning that it is possible

[19] By this, we mean that the rules may not perform operations which are so complicated that it cannot be checked by a computer program whether a given rule application is correct.

[20] There are exceptions to this. For instance, there is a number theoretical theorem known as Goodstein's theorem that can only be proven by using more powerful methods [Goodstein, 1944, Kirby and Paris, 1982].

$$\mathsf{succZero}\ \frac{\Gamma, \forall n.\, !\, zero \doteq succ(n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{succEq}\ \frac{\Gamma, \forall m.\forall n.(succ(m) \doteq succ(n) \mathrel{->} m \doteq n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{pluZero}\ \frac{\Gamma, \forall n.plus(zero, n) \doteq n \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{plusSucc}\ \frac{\Gamma, \forall m.\forall n.plus(succ(m), n) \doteq succ(plus(m, n)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{timesZero}\ \frac{\Gamma, \forall n.times(zero, n) \doteq zero \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{timesSucc}\ \frac{\Gamma, \forall m.\forall n.times(succ(m), n) \doteq plus(n, times(m, n)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

$$\mathsf{natInduct}\ \frac{\Gamma \Longrightarrow [n/zero](\phi), \Delta \qquad \Gamma \Longrightarrow \forall n.(\phi \mathrel{->} [n/succ(n)](\phi)), \Delta \quad \Gamma, \forall n.\phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

where ϕ is a formula with at most one free variable $n{:}N$.

Fig. 1.5. Rules for arithmetic, using variables $m{:}N, n{:}N$

to prove any valid statement about programs if one can prove statements about arithmetic. The observation about practically interesting statements applies also here, which means that despite the theoretical incompleteness, we can prove almost all interesting statements about almost all interesting programs.

References

Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. Springer Verlag, second edition, 2003.

Hans-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical logic*. Undergraduate texts in mathematics. Springer Verlag, 1984.

Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, 2000.

Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

Jean Gallier. *Logic for Computer Science*. Harper & Row Publisher, 1986. Revised online version from 2003 available from author's web page at `http://www.cis.upenn.edu/~jean/gbooks/logic.html`.

Martin Giese. A calculus for type predicates and type coercion. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, Tableaux 2005*, volume 3702 of *LNAI*, pages 123–137. Springer, 2005.

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38: 173–198, 1931.

Joseph Goguen and Jose Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

Reuben Louis Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944.

Jean Goubault-Larrecq and Ian Mackie. *Proof Theory and Automated Deduction*, volume 6 of *Applied Logic Series*. Kluwer Academic Publishers, May 1997.

Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, second edition, 2004.

Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bull. London. Math. Soc.*, 14:285–293, 1982.

Maria Manzano. *Extensions of First Order Logic*, volume 19 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996. Chapter VI and VII on many-sorted logic.

Anil Nerode and Richard A. Shore. *Logic for Applications*. Springer (second edition), 1979.

Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science B.V., 2001.