Chapter 2
First-Order Logic

Peter H. Schmitt

2.1 Introduction

The ultimate goal of first-order logic in the context of this book, and this applies to a great extent also to Computer Science in general, is the formalization of and reasoning with natural language specifications of systems and programs. This chapter provides the logical foundations for doing so in three steps. In Section 2.2 basic first-order logic (FOL) is introduced much in the tradition of Mathematical Logic as it evolved during the 20th century as a universal theory not tailored towards a particular application area. Already this section goes beyond what is usually found in textbooks on logic for computer science in that type hierarchies are included from the start. In the short Section 2.3 two features will be added to the basic logic, that did not interest the mathematical logicians very much but are indispensable for practical reasoning. In Section 2.4 the extended basic logic will be instantiated to Java first-order logic (JFOL), tailored for the particular task of reasoning about Java programs. The focus in the present chapter is on statements; programs themselves and formulas talking about more than one program state at once will enter the scene in Chapter 3.

2.2 Basic First-Order Logic

2.2.1 *Syntax*

Definition 2.1. A *type hierarchy* is a pair $\mathscr{T} = (\mathrm{TSym}, \sqsubseteq)$, where

1. TSym is a set of type symbols;
2. $\sqsubseteq$ is a reflexive, transitive relation on TSym, called the subtype relation;
3. there are two designated type symbols, the *empty* type $\bot \in \mathrm{TSym}$ and the *universal* type $\top \in \mathrm{TSym}$ with $\bot \sqsubseteq A \sqsubseteq \top$ for all $A \in \mathrm{TSym}$.

© Springer International Publishing AG 2016
W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 10001, pp. 23–47, 2016
DOI: 10.1007/978-3-319-49812-6_2

We point out that no restrictions are placed on type hierarchies in contrast to other approaches requiring the existence of unique lower bounds.

Two types A, B in $\mathscr{T}$ are called *incomparable* if neither $A \sqsubseteq B$ nor $B \sqsubseteq A$.

Definition 2.2. A *signature*, which is sometimes also called *vocabulary*, $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym})$ for a given type hierarchy $\mathscr{T}$ is made up of

1. a set FSym of typed function symbols,
   by $f : A_1 \times \ldots \times A_n \rightarrow A$ we declare the argument types of $f \in \mathrm{FSym}$ to be $A_1, \ldots, A_n$ in the given order and its result type to be A,
2. a set PSym of typed predicate symbols,
   by $p(A_1, \ldots, A_n)$ we declare the argument types of $p \in \mathrm{PSym}$ to be $A_1, \ldots, A_n$ in the given order,
   PSym obligatory contains the binary dedicated symbol $\doteq(\top, \top)$ for equality. and the two 0-place predicate symbols *true* and *false*.
3. a set VSym of typed variable symbols,
   by $v : A$ for $v \in \mathrm{VSym}$ we declare v to be a variable of type A.

All types A, A_i in this definition must be different from $\bot$. A 0-ary function symbol $c : \ \rightarrow A$ is called a constant symbol of type A. A 0-ary predicate symbol $p()$ is called a propositional variable or propositional atom. We do not allow overloading: The same symbol may not occur in $\mathrm{FSym} \cup \mathrm{PSym} \cup \mathrm{VSym}$ with different typing.

The next two definitions define by mutual induction the syntactic categories of terms and formulas of typed first-order logic.

Definition 2.3. Let $\mathscr{T}$ be a type hierarchy, and Σ a signature for $\mathscr{T}$. The set Trm_A of *terms of type* A, for $A \neq \bot$, is inductively defined by

1. $v \in \mathrm{Trm}_A$ for each variable symbol $v : A \in \mathrm{VSym}$ of type A.
2. $f(t_1, \ldots, t_n) \in \mathrm{Trm}_A$ for each $f : A_1 \times \ldots \times A_n \rightarrow A \in \mathrm{FSym}$ and all terms $t_i \in \mathrm{Trm}_{B_i}$ with $B_i \sqsubseteq A_i$ for $1 \leq i \leq n$.
3. $(\text{if } \phi \text{ then } t_1 \text{ else } t_2) \in \mathrm{Trm}_A$ for $\phi \in \mathrm{Fml}$ and $t_i \in \mathrm{Trm}_{A_i}$ such that $A_2 \sqsubseteq A_1 = A$ or $A_1 \sqsubseteq A_2 = A$.

If $t \in \mathrm{Trm}_A$ we say that t is of (static) type A and write $\alpha(t) = A$.

Note, that item (2) in Definition 3 entails $c \in \mathrm{Trm}_A$ for each constant symbol $c : \ \rightarrow A \in \mathrm{FSym}$. Since we do not allow overloading there is for every term only one type A with $t \in \mathrm{Trm}_A$. This justifies the use of the function symbol α.

Terms of the form defined in item (3) are called *conditional terms*. They are a mere convenience. For every formula with conditional terms there is an equivalent formula without them. More liberal typing rules are possible. The theoretically most satisfying solution would be to declare the type of $(\text{if } \phi \text{ then } t_1 \text{ else } t_2)$ to be the least common supertype $A_1 \sqcup A_2$ of A_1 and A_2. But, the assumption that $A_1 \sqcup A_2$ always exists would lead to strange consequences in the program verification setting.

Definition 2.4. The set Fml of *formulas* of first-order logic for a given type hierarchy $\mathscr{T}$ and signature Σ is inductively defined as:

1. $p(t_1,\ldots,t_n) \in \mathrm{Fml}$ for $p(A_1,\ldots,A_n) \in \mathrm{PSym}$, and $t_i \in \mathrm{Trm}_{B_i}$ with $B_i \sqsubseteq A_i$ for all $1 \leq i \leq n$.
   As a consequence of item 2 in Definition 2.2 we know
   $t_1 \doteq t_2 \in \mathrm{Fml}$ for arbitrary terms t_i and *true* and *false* are in Fml.
2. $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \to \psi)$, $(\phi \leftrightarrow \psi)$ are in Fml for arbitrary $\phi, \psi \in \mathrm{Fml}$.
3. $\forall v;\phi$, $\exists v;\phi$ are in Fml for $\phi \in \mathrm{Fml}$ and $v : A \in \mathrm{VSym}$.

As an inline footnote we remark that the notation for conditional terms can also be used for formulas. The *conditional formula* (if ϕ_1 then ϕ_2 else ϕ_3) is equivalent to $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \phi_3)$.

If need arises we will make dependence of these definitions on Σ and $\mathscr{T}$ explicit by writing $\mathrm{Trm}_{A,\Sigma}$, Fml_Σ or $\mathrm{Trm}_{A,\mathscr{T},\Sigma}$, $\mathrm{Fml}_{\mathscr{T},\Sigma}$. When convenient we will also use the redundant notation $\forall\, A\ v;\phi$, $\exists\, A\ v;\phi$ for a variable $v : A \in \mathrm{VSym}$.

Formulas built by clause (1) only are called *atomic formulas*.

Definition 2.5. For terms t and formulas ϕ we define the sets $var(t)$, $var(\phi)$ of all variables occurring in t or ϕ and the sets $fv(t), fv(\phi)$ of all variables with at least one free occurrence in t or ϕ:

$$\begin{array}{llllr}
var(v) = & \{v\} & fv(v) = & \{v\} & \text{for } v \in \mathrm{VSym} \\
var(t) = & \bigcup_{i=1}^{n} var(t_i) & fv(t) = & \bigcup_{1=i}^{n} fv(t_i) & \text{for } t = f(t_1,\ldots,t_n) \\
var(t) = & var(\phi) \cup & fv(t) = & fv(\phi) \cup & \text{for } t = \\
 & var(t_1) \cup var(t_2) & & fv(t_1) \cup fv(t_2) & (\text{if } \phi \text{ then } t_1 \text{ else } t_2) \\
var(\phi) = & \bigcup_{i=1}^{n} var(t_i) & fv(\phi) = & \bigcup_{i=1}^{n} fv(t_i) & \text{for } \phi = p(t_1,\ldots,t_n) \\
var(\neg\phi) = & var(\phi) & fv(\neg\phi) = & fv(\phi) & \\
var(\phi) = & var(\phi_1) \cup var(\phi_2) & fv(\phi) = & fv(\phi_1) \cup fv(\phi_2) & \text{for } \phi = \phi_1 \circ \phi_2 \\
 & & & \multicolumn{2}{r}{\text{where } \circ \text{ is any binary Boolean operation}} \\
\multicolumn{2}{l}{var(Q\ v.\phi) = var(\phi)} & \multicolumn{2}{l}{fv(Q\ v.\phi) = var(\phi) \setminus \{v\}} & \text{where } Q \in \{\forall, \exists\}
\end{array}$$

A term without free variables is called a *ground term*, a formula without free variables a *ground formula* or *closed formula*.

It is an obvious consequence of this definition that every occurrence of a variable v in a term or formula with empty set of free variables is within the scope of a quantifier $Q\ v$.

One of the most important syntactical manipulations of terms and formulas are substitutions, that replace variables by terms. They will play a crucial role in proofs of quantified formulas as well as equations.

Definition 2.6. A *substitution* τ is a function that associates with every variable v a type compatible term $\tau(v)$, i.e., if v is of type A then $\tau(v)$ is a term of type A' such that $A' \sqsubseteq A$.

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

A substitution τ is called a *ground substitution* if $\tau(v)$ is a ground term for all $v \in \mathrm{dom}(\tau)$.

We will only encounter substitutions τ such that $\tau(v) = v$ for all but finitely many variables v. The set $\{v \in \mathrm{VSym} \mid \tau(v) \neq v\}$ is called the *domain* of τ. It remains to make precise how a substitution τ is applied to terms and formulas.

Definition 2.7. Let τ be a substitution and t a term, then $\tau(t)$ is recursively defined by:

1. $\tau(x) = x$ if $x \notin \mathrm{dom}(\tau)$
2. $\tau(x)$ as in the definition of τ if $x \in \mathrm{dom}(\tau)$
3. $\tau(f(t_1,\ldots,t_k)) = f(\tau(t_1),\ldots,\tau(t_k))$ if $t = f(t_1,\ldots,t_k)$

Let τ be a ground substitution and ϕ a formula, then $\tau(\phi)$ is recursive defined

4. $\tau(\mathit{true}) = \mathit{true}$, $\tau(\mathit{false}) = \mathit{false}$
5. $\tau(p(t_1,\ldots,t_k)) = p(\tau(t_1),\ldots,\tau(t_k))$ if ϕ is the atomic formula $p(t_1,\ldots,t_k)$
6. $\tau(t_1 \doteq t_k) = \tau(t_1) \doteq \tau(t_k)$
7. $\tau(\neg\phi) = \neg\tau(\phi)$
8. $\tau(\phi_1 \circ \phi_2) = \tau(\phi_1) \circ \tau(\phi_2)$ for propositional operators $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
9. $\tau(Qv.\phi) = Qv.\tau_v(\phi)$ for $Q \in \{\exists, \forall\}$ and $\mathrm{dom}(\tau_v) = \mathrm{dom}(\tau) \setminus \{v\}$ with $\tau_v(x) = \tau(x)$ for $x \in \mathrm{dom}(\tau_v)$.

There are some easy conclusions from these definitions:

- If $t \in \mathrm{Trm}_A$ then $\tau(t)$ is a term of type A' with $A' \sqsubseteq A$. Indeed, if t is not a variable then $\tau(t)$ is again of type A.
- $\tau(\phi)$ meets the typing restrictions set forth in Definition 2.4.

Item 9 deserves special attention. Substitutions only act on free variables. So, when computing $\tau(Qv.\phi)$, the variable v in the body ϕ of the quantified formula is left untouched. This is effected by removing v from the domain of τ.

It is possible, and quite common, to define also the application of nonground substitutions to formulas. Care has to be taken in that case to avoid *clashes*, see Example 2.8 below. We will only need ground substitutions later on, so we sidestep this difficulty.

Example 2.8. For the sake of this example we assume that there is a type symbol *int* $\in$ TSym, function symbols $+ : \mathit{int} \times \mathit{int} \rightarrow \mathit{int}$, $* : \mathit{int} \times \mathit{int} \rightarrow \mathit{int}$, $- : \mathit{int} \rightarrow \mathit{int}$, $\mathit{exp} : \mathit{int} \times \mathit{int} \rightarrow \mathit{int}$ and constants $0 : \mathit{int}$, $1 : \mathit{int}$, $2 : \mathit{int}$, in FSym. Definition 2.3 establishes an abstract syntax for terms. In examples we are free to use a concrete, or pretty-printing syntax. Here we use the familiar notation $a+b$ instead of $+(a,b)$, $a*b$ or ab instead of $*(a,b)$, and a^b instead of $\mathit{exp}(a,b)$. Let furthermore $x : \mathit{int}$, $y : \mathit{int}$ be variables of sort *int*. The following table shows the results of applying the substitution $\tau_1 = [x/0, y/1]$ to the given formulas

$$\begin{array}{lll}
\phi_1 = \forall x; ((x+y)^2 \doteq x^2 + 2xy + y^2) & \tau_1(\phi_1) = \forall x; ((x+1)^2 \doteq x^2 + 2*x*1 + 1^2) \\
\phi_2 = (x+y)^2 \doteq x^2 + 2xy + y^2 & \tau_1(\phi_2) = (0+1)^2 \doteq 0^2 + 2*0*1 + 1^2 \\
\phi_3 = \exists x; (x > y) & \tau_1(\phi_3) = \exists x; (x > 1)
\end{array}$$

Application of the nonground substitution $\tau_2 = [y/x]$ on ϕ_3 leads to $\exists x; (x > x)$. While $\exists x; (x > y)$ is true for all assignments to y the substituted formula $\tau(\phi_3)$ is not.

Validity is preserved if we restrict to clash-free substitutions. A substitution τ is said to create a *clash* with formula ϕ if a variable w in a term $\tau(v)$ for $v \in \mathrm{dom}(\tau)$ ends up in the scope of a quantifier Qw in ϕ. For τ_2 the variable x in $\tau_2(y)$ will end up in the scope of $\forall x$;

The concept of a substitution also comes in handy to solve the following notational problem. Let ϕ be a formula that contains somewhere an occurrence of the term t_1. How should we refer to the formula arising from ϕ by replacing t_1 by t_2? E.g. replace $2xy$ in ϕ_2 by $xy2$. The solution is to use a new variable z and a formula ϕ_0 such that $\phi = [z/t_1]\phi_0$. Then the replaced formula can be referred to as $[z/t_2]\phi_0$. In the example we would have $\phi_0 = (x+y)^2 \doteq x^2 + z + y^2$. This trick will be extensively used in Figure 2.1 and 2.2.

2.2.2 Calculus

The main reason nowadays for introducing a formal, machine readable syntax for formulas, as we did in the previous subsection, is to get machine support for logical reasoning. For this, one needs first a suitable calculus and then an efficient implementation. In this subsection we present the rules for basic first-order logic. A machine readable representation of these rules will be covered in Chapter 4. Chapter 15 provides an unhurried introduction on using the KeY theorem prover based on these rules that can be read without prerequisites. So the reader may want to step through it before continuing here.

The calculus of our choice is the *sequent calculus*. The basic data that is manipulated by the rules of the sequent calculus are *sequents*. These are of the form $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$. The formulas $\phi_1, \ldots, \phi_n$ at the left-hand side of the sequent separator $\Longrightarrow$ are the antecedents of the sequent; the formulas $\psi_1, \ldots, \psi_m$ on the right are the succedents. In our version of the calculus antecedent and succedent are sets of formulas, i.e., the order and multiple occurrences are not relevant. Furthermore, we will assume that all ϕ_i and ψ_j are ground formulas. A sequent $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$ is valid iff the formula $\bigwedge_{1=i}^{n} \phi_i \to \bigvee_{1=j}^{m} \psi_j$ is valid.

The concept of sequent calculi was introduce by the German logician Gerhard Gentzen in the 1930s, though for a very different purpose.

Figures 2.1 and 2.2 show the usual set of rules of the sequent calculus with equality as it can be found in many text books, e.g. [Gallier, 1987, Section 5.4]. Rules are written in the form

$$\mathsf{ruleName}\ \frac{P_1, \ldots P_n}{C}$$

The P_i is called the *premisses* and C the *conclusion* of the rule. There is no theoretical limit on n, but most of the time $n = 1$, sometimes $n = 2$, and in rare cases $n = 3$. Note, that premiss and conclusion contain the schematic variables Γ, Δ for set of formulas, ψ, ϕ for formulas and t, c for terms and constants. We use Γ, ϕ and ψ, Δ to stand for $\Gamma \cup \{\phi\}$ and $\{\psi\} \cup \Delta$. An instance of a rule is obtained by consistently replacing the

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

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

$$\mathsf{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi, \Delta} \qquad \mathsf{impLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Longrightarrow \Delta}$$

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

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow [x/c](\phi), \Delta}{\Gamma \Longrightarrow \forall x; \phi, \Delta} \qquad \mathsf{allLeft}\ \frac{\Gamma, \forall x; \phi, [x/t](\phi) \Longrightarrow \Delta}{\Gamma, \forall x; \phi \Longrightarrow \Delta}$$

with $c : \rightarrow A$ a new constant, if $x{:}A$ with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if $x{:}A$

$$\mathsf{exLeft}\ \frac{\Gamma, [x/c](\phi) \Longrightarrow \Delta}{\Gamma, \exists x; \phi \Longrightarrow \Delta} \qquad \mathsf{exRight}\ \frac{\Gamma \Longrightarrow \exists x; \phi, [x/t](\phi), \Delta}{\Gamma \Longrightarrow \exists x; \phi, \Delta}$$

with $c : \rightarrow A$ a new constant, if $x{:}A$ with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if $x{:}A$

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

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

Figure 2.1 First-order rules for the logic FOL

schematic variables in premiss and conclusion by the corresponding entities: sets of formulas, formulas, etc. Rule application in KeY proceeds from bottom to top. Suppose we want to prove a sequent s_2. We look for a rule R such that there is an instantiation *Inst* of the schematic variables in R such that the instantiation of its conclusion $Inst(S_2)$ equals s_2. After rule application we are left with the task to prove the sequent $Inst(S_1)$. If S_1 is empty, we succeeded.

Definition 2.9. The rules close, closeFalse, and closeTrue from Figure 2.1 are called *closing rules* since their premisses are empty.

Since there are rules with more than one premiss the proof process sketched above will result in a proof tree.

Definition 2.10. A *proof tree* is a tree, shown with the root at the bottom, such that

1. each node is labeled with a sequent or the symbol $*$,
2. if an inner node n is annotated with $\Gamma \Longrightarrow \Delta$ then there is an instance of a rule whose conclusion is $\Gamma \Longrightarrow \Delta$ and the child node, or children nodes of n are labeled with the premiss or premisses of the rule instance.

A branch in a proof tree is called *closed* if its leaf is labeled by $*$. A proof tree is called *closed* if all its branches are closed, or equivalently if all its leaves are labeled with $*$.

We say that a sequent $\Gamma \Longrightarrow \Delta$ can be derived if there is a closed proof tree whose root is labeled by $\Gamma \Longrightarrow \Delta$.

As a first simple example, we will derive the sequent $\Longrightarrow p \land q \to q \land p$. The same formula is also used in the explanation of the KeY prover in Chapter 15. As its antecedent is empty, this sequent says that the propositional formula $p \land q \to q \land p$ is a tautology. Application of the rule impRight reduces our proof goal to $p \land q \Longrightarrow q \land p$ and application of andLeft further to $p, q \Longrightarrow q \land p$. Application of andRight splits the proof into the two goals $p, q \Longrightarrow q$ and $p, q \Longrightarrow p$. Both goals can be discharged by an application of the close rule. The whole proof can concisely be summarized as a tree

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{p,q \Longrightarrow q} \qquad \dfrac{*}{p,q \Longrightarrow p}}{p,q \Longrightarrow q \land p}}{p \land q \Longrightarrow q \land p}}{\Longrightarrow p \land q \to q \land p}$$

Let us look at an example derivation involving quantifiers. If you are puzzled by the use of substitutions $[x/t]$ in the formulations of the rules you should refer back to Example 2.8. We assume that $p(A,A)$ is a binary predicate symbol with both arguments of type A. Here is the, nonbranching, proof tree for the formula $\exists v; \forall w; p(v,w) \to \forall w; \exists v; p(v,w)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\forall w; p(c,w), p(c,d) \Longrightarrow p(c,d), \exists v; p(v,d)}}{\forall w; p(c,w) \Longrightarrow \exists v; p(v,d)}}{\exists v; \forall w; p(v,w) \Longrightarrow \forall w; \exists v; p(v,w)}}{\Longrightarrow \exists v; \forall w; p(v,w) \to \forall w; \exists v; p(v,w)}$$

The derivation starts, from bottom to top, with the rule impRight. The next line above is obtained by applying exLeft and allRight. This introduces new constant symbols $c :\to A$ and $d :\to A$. The top line is obtained by the rules exRight and allLeft with the ground substitutions $[w/d]$ and $[v/c]$. The proof terminates by an application of close resulting in an empty proof obligation. An application of the rules exLeft, allRight is often called *Skolemization* and the new constant symbols called *Skolem constants*. The rules involving equality are shown in Figure 2.2. The rules eqLeft and eqRight formalize the intuitive application of equations: if $t_1 \doteq t_2$ is known, we may replace wherever we want t_1 by t_2. In typed logic the formula after substitution might not be well-typed. Here is an example for the rule eqLeft without restriction. Consider two types $A \neq B$ with $B \sqsubseteq A$, two constant symbols $a : \to A$ and $b : \to B$, and a unary predicate $p(B)$. Applying unrestricted eqLeft on the sequent $b \doteq a, p(b) \Longrightarrow$ would result in $b \doteq a, p(b), p(a) \Longrightarrow$. There is in a sense logically nothing wrong with this, but $p(a)$ is not well-typed. This motivates the provisions in the rules eqLeft and eqRight.

$$\mathsf{eqLeft}\ \frac{\Gamma, t_1 \doteq t_2, [z/t_1](\phi), [z/t_2](\phi) \Longrightarrow \Delta}{\Gamma, t_1 \doteq t_2, [z/t_1](\phi) \Longrightarrow \Delta}$$
provided $[z/t_2](\phi)$ is well-typed

$$\mathsf{eqRight}\ \frac{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_2](\phi), [z/t_1](\phi), \Delta}{\Gamma, t_1 \doteq t_2 \Longrightarrow [z/t_1](\phi), \Delta}$$
provided $[z/t_2](\phi)$ is well-typed

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

Figure 2.2 Equality rules for the logic FOL

Let us consider a short example of equational reasoning involving the function symbol $+ : \mathit{int} \times \mathit{int} \to \mathit{int}$.

7 $*$

6 $(a+(b+c))+d) \doteq a+((b+c)+d), \forall x,y,z; ((x+y)+z \doteq x+(y+z))$
$(b+c)+d \doteq b+(c+d), a+(b+c))+d) \doteq a+(b+(c+d)) \Longrightarrow$
$(a+(b+c))+d) \doteq a+(b+(c+d))$

5 $(a+(b+c))+d) \doteq a+((b+c)+d), \forall x,y,z; ((x+y)+z \doteq x+(y+z))$
$(b+c)+d \doteq b+(c+d) \Longrightarrow$
$(a+(b+c))+d) \doteq a+(b+(c+d))$

4 $(a+(b+c))+d) \doteq a+((b+c)+d), \forall x,y,z; ((x+y)+z \doteq x+(y+z)) \Longrightarrow$
$(a+(b+c))+d) \doteq a+(b+(c+d))$

3 $\forall x,y,z; ((x+y)+z \doteq x+(y+z)) \Longrightarrow (a+(b+c))+d \doteq a+(b+(c+d))$

2 $\forall x,y,z; ((x+y)+z \doteq x+(y+z)) \Longrightarrow$
$\forall x,y,z,u; (((x+(y+z))+u) \doteq x+(y+(z+u)))$

1 $\Longrightarrow \forall x,y,z; ((x+y)+z \doteq x+(y+z)) \to$
$\forall x,y,z,u; (((x+(y+z))+u) \doteq x+(y+(z+u)))$

Line 1 states the proof goal, a consequence from the associativity of $+$. Line 2 is obtained by an application of impRight while line 3 results from a four-fold application of allRight introducing the new constant symbol a, b, c, d for the universally quantified variables x, y, z, u, respectively. Line 4 in turn is arrived at by an application of allLeft with the substitution $[x/a, y/(b+c), z/d]$. Note, that the universally quantified formula does not disappear. In Line 5 another application of allLeft, but this time with the substitution $[x/b, y/c, z/d]$, adds the equation $(b+c)+d \doteq b+(c+d)$ to the antecedent. Now, eqLeft is applicable, replacing on the left-hand side of the sequent the term $(b+c)+d$ in $(a+b)+(c+d) \doteq a+(b+(c+d))$ by the right-hand side of the equation $(b+c)+d \doteq b+(c+d)$. This results in the same equation as in the succedent. Rule close can thus be applied.

Already this small example reveals the technical complexity of equational reasoning. Whenever the terms involved in equational reasoning are of a special type one would prefer to use decision procedures for the relevant specialized theories, e.g., for integer arithmetic or the theory of arrays.

We will see in the next section, culminating in Theorem 2.20, that the rules from Figures 2.1 and 2.2 are sufficient with respect to the semantics to be introduced in that section. But, it would be very inefficient to base proofs only on these first principles. The KeY system contains many derived rules to speed up the proof process. Let us just look at one randomly chosen example:

$$\mathsf{doubleImpLeft}\ \frac{\Gamma \Longrightarrow b, \Delta \qquad \Gamma \Longrightarrow c, \Delta \qquad \Gamma, d \Longrightarrow \Delta}{\Gamma, b \to (c \to d) \Longrightarrow \Delta}$$

It is easy to see that doubleImpLeft can be derived.

There is one more additional rule that we should not fail to mention:

$$\mathsf{cut}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

provided ϕ is a ground formula

On the basis of the notLeft rule this is equivalent to

$$\mathsf{cut}'\ \frac{\Gamma, \neg\phi \Longrightarrow \Delta \qquad \Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

provided ϕ is a ground formula

It becomes apparent that the cut rule allows at any node in the proof tree proceeding by a case distinction. This is the favorite rule for user interaction. The system might not find a proof for $\Gamma \Longrightarrow \Delta$ automatically, but for a cleverly chosen ϕ automatic proofs for both $\Gamma, \phi \Longrightarrow \Delta$ and $\Gamma \Longrightarrow \phi, \Delta$ might be possible.

2.2.3 Semantics

So far we trusted that the logical rules contained in Figures 2.1 and 2.2 are self-evident. In this section we provide further support that the rules and the deduction system as a whole are sound, in particular no contradiction can be derived. So far we also had only empirical evidence that the rules are sufficient. The semantical approach presented in this section will open up the possibility to rigorously prove completeness.

Definition 2.11. A *universe* or *domain* for a given type hierarchy $\mathscr{T}$ and signature Σ consists of

1. a set D,

2. a typing function $\delta : D \to \mathrm{TSym} \setminus \{\perp\}$ such that for every $A \in \mathrm{TSym}$ the set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ is not empty.

The set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ is called the type universe or type domain for A. Definition 2.11 implies that for different types $A, B \in \mathrm{TSym} \setminus \{\perp\}$ there is an element $o \in D^A \cap D^B$ only if there exists $C \in \mathrm{TSym}$, $C \neq \perp$ with $C \sqsubseteq A$ and $C \sqsubseteq B$.

Lemma 2.12. *The type domains for a universe (D, δ) share the following properties*

1. *$D^\perp = \emptyset$, $D^\top = D$,*
2. *$D^A \subseteq D^B$ if $A \sqsubseteq B$,*
3. *$D^C = D^A \cap D^B$ in case the greatest lower bound C of A and B exists.*

Definition 2.13. A first-order *structure* $\mathscr{M}$ for a given type hierarchy $\mathscr{T}$ and signature Σ consists of

- a domain (D, δ),
- an interpretation I

such that

1. $I(f)$ is a function from $D^{A_1} \times \cdots \times D^{A_n}$ into D^A for $f : A_1 \times \ldots \times A_n \to A$ in FSym,
2. $I(p)$ is a subset of $D^{A_1} \times \cdots \times D^{A_n}$ for $p(A_1, \ldots, A_n)$ in PSym,
3. $I(\doteq) = \{(d, d) \mid d \in D\}$.

For constant symbols $c : \ \to A \in \mathrm{FSym}$ requirement (1) reduces to $I(c) \in D^A$. It has become customary to interpret an empty product as the set $\{\emptyset\}$, where $\emptyset$ is deemed to stand for the empty tuple. Thus requirement (2) reduces for $n = 0$ to $I(p) \subseteq \{\emptyset\}$. Only if need arises, we will say more precisely that $\mathscr{M}$ is a $\mathscr{T}$-Σ-structure.

Definition 2.14. Let $\mathscr{M}$ be a first-order structure with universe D.

A *variable assignment* is a function $\beta : \mathrm{VSym} \to D$ such that $\beta(v) \in D^A$ for $v : A \in \mathrm{VSym}$.

For a variable assignment β, a variable $v : A \in \mathrm{VSym}$ and a domain element $d \in D^A$, the following definition of a modified assignment will be needed later on:

$$\beta_v^d(v') = \begin{cases} d & \text{if } v' = v \\ \beta(v') & \text{if } v' \neq v \end{cases}$$

The next two definitions define the evaluation of terms and formulas with respect to a structure $\mathscr{M} = (D, \delta, I)$ for given type hierarchy $\mathscr{T}$, signature Σ, and variable assignment β by mutual recursion.

Definition 2.15. For every term $t \in \mathrm{Trm}_A$, we define its evaluation $\mathrm{val}_{\mathscr{M},\beta}(t)$ inductively by:

- $\mathrm{val}_{\mathscr{M},\beta}(v) = \beta(v)$ for any variable v.

- $\mathrm{val}_{\mathcal{M},\beta}(f(t_1,\ldots,t_n)) = I(f)(\mathrm{val}_{\mathcal{M},\beta}(t_1),\ldots,\mathrm{val}_{\mathcal{M},\beta}(t_n))$.
- $\mathrm{val}_{\mathcal{M},\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2) = \begin{cases} \mathrm{val}_{\mathcal{M},\beta}(t_1) \text{ if } (\mathcal{M},\beta) \models \phi \\ \mathrm{val}_{\mathcal{M},\beta}(t_2) \text{ if } (\mathcal{M},\beta) \not\models \phi \end{cases}$

Definition 2.16. For every formula $\phi \in \mathrm{Fml}$, we define when ϕ is considered to be true with respect to $\mathcal{M}$ and β, which is denoted with $(\mathcal{M},\beta) \models \phi$, by:

$$\begin{array}{lll}
1 & (\mathcal{M},\beta) \models \mathit{true},\ \ (\mathcal{M},\beta) \not\models \mathit{false} & \\
2 & (\mathcal{M},\beta) \models p(t_1,\ldots,t_n) & \text{iff } (\mathrm{val}_{\mathcal{M},\beta}(t_1),\ldots,\mathrm{val}_{\mathcal{M},\beta}(t_n)) \in I(p) \\
3 & (\mathcal{M},\beta) \models \neg\phi & \text{iff } (\mathcal{M},\beta) \not\models \phi \\
4 & (\mathcal{M},\beta) \models \phi_1 \land \phi_2 & \text{iff } (\mathcal{M},\beta) \models \phi_1 \text{ and } (\mathcal{M},\beta) \models \phi_2 \\
5 & (\mathcal{M},\beta) \models \phi_1 \lor \phi_2 & \text{iff } (\mathcal{M},\beta) \models \phi_1 \text{ or } (\mathcal{M},\beta) \models \phi_2 \\
6 & (\mathcal{M},\beta) \models \phi_1 \to \phi_2 & \text{iff } (\mathcal{M},\beta) \not\models \phi_1 \text{ or } (\mathcal{M},\beta) \models \phi_2 \\
7 & (\mathcal{M},\beta) \models \phi_1 \leftrightarrow \phi_2 & \text{iff } ((\mathcal{M},\beta) \models \phi_1 \text{ and } (\mathcal{M},\beta) \models \phi_2) \text{ or} \\
 & & \text{iff } ((\mathcal{M},\beta) \not\models \phi_1 \text{ and } (\mathcal{M},\beta) \not\models \phi_2) \\
8 & (\mathcal{M},\beta) \models \forall\, A\ v;\phi & \text{iff } (\mathcal{M},\beta_v^d) \models \phi \text{ for all } d \in D^A \\
9 & (\mathcal{M},\beta) \models \exists\, A\ v;\phi & \text{iff } (\mathcal{M},\beta_v^d) \models \phi \text{ for at least one } d \in D^A
\end{array}$$

For a 0-place predicate symbol p, clause (2) says $\mathcal{M} \models p$ iff $\emptyset \in I(p)$. Thus the interpretation I acts in this case as an assignment of truth values to p. This explains why we have called 0-place predicate symbols propositional atoms.

Given the restriction on $I(\doteq)$ in Definition 2.13, clause (2) also says $(\mathcal{M},\beta) \models t_1 \doteq t_2$ iff $\mathrm{val}_{\mathcal{M},\beta}(t_1) = \mathrm{val}_{\mathcal{M},\beta}(t_2)$.

For a set Φ of formulas, we use $(\mathcal{M},\beta) \models \Phi$ to mean $(\mathcal{M},\beta) \models \phi$ for all $\phi \in \Phi$.

If ϕ is a formula without free variables, we may write $\mathcal{M} \models \phi$ since the variable assignment β is not relevant here.

To prepare the ground for the next definition we explain the concept of extensions between type hierarchies.

Definition 2.17. A type hierarchy $\mathcal{T}_2 = (\mathrm{TSym}_2, \sqsubseteq_2)$ is an *extension* of a type hierarchy $\mathcal{T}_1 = (\mathrm{TSym}_1, \sqsubseteq_1)$, in symbols $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$, if

1. $\mathrm{TSym}_1 \subseteq \mathrm{TSym}_2$
2. $\sqsubseteq_2$ is the smallest subtype relation containing $\sqsubseteq_1 \cup \Delta$ where Δ is a set of pairs (S,T) with $T \in \mathrm{TSym}_1$ and $S \in \mathrm{TSym}_2 \setminus \mathrm{TSym}_1$.

So, new types can only be declared to be subtypes of old types, never supertypes. Also, $\bot \sqsubseteq_2 A \sqsubseteq_2 \top$ for all new types A.

Definition 2.17 forbids the introduction of subtype chains like $A \sqsubseteq B \sqsubseteq T$ into the type hierarchy. However, it can be shown that relaxing the definition in that respect results in an equivalent notion of logical consequence. We keep the restriction here since it simplifies reasoning about type hierarchy extensions.

For later reference, we note the following lemma.

Lemma 2.18. *Let $\mathcal{T}_2 = (\mathrm{TSym}_2, \sqsubseteq_2)$ be an extension of $\mathcal{T}_1 = (\mathrm{TSym}_1, \sqsubseteq_1)$ with $\sqsubseteq_2$ the smallest subtype relation containing $\sqsubseteq_1 \cup \Delta$, for some $\Delta \subseteq (\mathrm{TSym}_2 \setminus \mathrm{TSym}_1) \times \mathrm{TSym}_1$.*

Then, for $A, B \in \mathrm{TSym}_1$, $C \in \mathrm{TSym}_2 \setminus \mathrm{TSym}_1$, $D \in \mathrm{TSym}_2$

1. $A \sqsubseteq_2 B$ *iff* $A \sqsubseteq_1 B$
2. $C \sqsubseteq_2 A$ *iff* $T \sqsubseteq_1 A$ *for some* $(C,T) \in \Delta$.
3. $D \sqsubseteq_2 C$ *iff* $D = C$ *or* $D = \bot$

Proof. This follows easily from the fact that no supertype relations of the form $A \sqsubseteq_2 C$ for new type symbols C are stipulated. $\square$

Definition 2.19. Let $\mathscr{T}$ be a type hierarchy and Σ a signature, $\phi \in \mathrm{Fml}_{\mathscr{T},\Sigma}$ a formula without free variables, and $\Phi \subseteq \mathrm{Fml}_{\mathscr{T},\Sigma}$ a set of formulas without free variables.

1. ϕ is a *logical consequence* of Φ, in symbols $\Phi \models \phi$, if for all type hierarchies $\mathscr{T}'$ with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and all $\mathscr{T}'$-Σ-structures $\mathscr{M}$ such that $\mathscr{M} \models \Phi$, also $\mathscr{M} \models \phi$ holds.
2. ϕ is *universally valid* if it is a logical consequence of the empty set, i.e., if $\emptyset \models \phi$.
3. ϕ is satisfiable if there is a type hierarchy $\mathscr{T}'$, with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and a $\mathscr{T}'$-Σ-structure $\mathscr{M}$ with $\mathscr{M} \models \phi$.

The extension of Definition 2.19 to formulas with free variables is conceptually not difficult but technically a bit involved. The present definition covers however all we need in this book.

The central concept is universal validity since, for finite Φ, it can easily be seen that:

- $\Phi \models \phi$ iff the formula $\bigwedge \Phi \rightarrow \phi$ is universally valid.
- ϕ is satisfiable iff $\neg\phi$ is not universally valid.

The notion of *logical consequence* from Definition 2.19 is sometimes called *super logical consequence* to distinguish it from the concept $\Phi \models_{\mathscr{T},\Sigma} \phi$ denoting that for any $\mathscr{T}$-Σ-structure $\mathscr{M}$ with $\mathscr{M} \models \Phi$ also $\mathscr{M} \models \phi$ is true.

To see the difference, let the type hierarchy $\mathscr{T}_1$ contain types A and B such that the greatest lower bound of A and B is $\bot$. For the formula $\phi_1 = \forall\, A\, x; (\forall\, B\, y; (x \neq y))$ we have $\models_{\mathscr{T}_1} \phi_1$. Let $\mathscr{T}_2$ be the type hierarchy extending $\mathscr{T}_1$ by a new type D and the ordering $D \sqsubseteq A$, $D \sqsubseteq B$. Now, $\models_{\mathscr{T}_2} \phi_1$ does no longer hold true.

The phenomenon that the tautology property of a formula ϕ depends on symbols that do not occur in ϕ is highly undesirable. This is avoided by using the logical consequence defined as above. In this case we have $\not\models \phi_1$.

Theorem 2.20 (Soundness and Completeness Theorem). *Let $\mathscr{T}$ be a type hierarchy and Σ a signature, $\phi \in \mathrm{Fml}_{\mathscr{T},\Sigma}$ without free variables. The calculus for FOL is given by the rules in Figures 2.1 and 2.2. Assume that for every type $A \in \mathscr{T}$ there is a constant symbol of type A' with $A' \sqsubseteq A$.*

Then:

- *if there is a closed proof tree in FOL for the sequent* $\Longrightarrow \phi$ *then* ϕ *is universally valid*
  i.e., FOL is sound.
- *if* ϕ *is universally valid then there is a closed proof tree for the sequent* $\Longrightarrow \phi$ *in FOL.*
  i.e., FOL is complete.

For the untyped calculus a proof of the sound- and completeness theorem may be found in any decent text book, e.g. [Gallier, 1987, Section 5.6]. Giese [2005] covers the typed version in a setting with additional cast functions and type predicates. His proof does not consider super logical consequence and requires that type hierarchies are lower-semi-lattices.

Concerning the constraint placed on the signature in Theorem 2.20, the calculus implemented in the KeY system takes a slightly different but equivalent approach: instead of requiring the existence of sufficient constants, it allows one to derive via the rule ex_unused, for every $A \in \mathscr{T}$ the formula $\exists x(x \doteq x)$, with x a variable of type A.

Definition 2.21. A rule

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \qquad \Gamma_2 \Longrightarrow \Delta_2}{\Gamma \Longrightarrow \Delta}$$

of a sequent calculus is called

- *sound* if whenever $\Gamma_1 \Longrightarrow \Delta_1$ and $\Gamma_2 \Longrightarrow \Delta_2$ are universally valid so is $\Gamma \Longrightarrow \Delta$.
- *complete* if whenever $\Gamma \Longrightarrow \Delta$ is universally valid then also $\Gamma_1 \Longrightarrow \Delta_1$ and $\Gamma_2 \Longrightarrow \Delta_2$ are universally valid.

For nonbranching rules and rules with side conditions the obvious modifications have to be made.

An inspection of the proof of Theorem 2.20 shows that if all rules of a calculus are sound then the calculus itself is sound. This is again stated as Lemma 4.7 in Section 4.4 devoted to the soundness management of the KeY system. In the case of soundness also the reverse implication is true: if a calculus is sound then all its rules will be sound.

The inspection of the proof of Theorem 2.20 also shows that the calculus is complete if all its rules are complete. This criterion is however not necessary, a complete calculus may contain rules that are not complete.

2.3 Extended First-Order Logic

In this section we extend the Basic First-Order Logic from Section 2.2. First we turn our attention in Subsection 2.3.1 to an additional term building construct: *variable binders*. They do not increase the expressive power of the logic, but are extremely handy.

An issue that comes up in almost any practical use of logic, are partial functions. In the KeY system, partial functions are treated via underspecification as explained in Subsection 2.3.2. In essence this amounts to replacing a partial function by all its extensions to total functions.

2.3.1 Variable Binders

This subsection assumes that the type *int* of mathematical integers, the type *LocSet* of sets of locations, and the type *Seq* of finite sequences are present in TSym. For the logic JFOL to be presented in Subsection 2.4 this will be obligatory.

A typical example of a variable binder symbol is the sum operator, as in $\Sigma_{k=1}^{n} k^2$. Variable binders are related to quantifiers in that they *bind* a variable. The KeY system does not provide a generic mechanism to include new binder symbols. Instead we list the binder symbols included at the moment.

A more general account of binder symbols is contained in the doctoral thesis [Ulbrich, 2013, Subsection 2.3.1]. Binder symbols do not increase the expressive power of first-order logic: for any formula ϕ_b containing binder symbols there is a formula ϕ without such that ϕ_b is universally valid if and only if ϕ is, see [Ulbrich, 2013, Theorem 2.4]. This is the reason why one does not find binder symbols other than quantifiers in traditional first-order logic text books.

Definition 2.22 (extends Definition 2.3).

4. If vi is a variable of type *int*, b_0, b_1 are terms of type *int* not containing vi and s is an arbitrary term in Trm_{int}, then $bsum\{vi\}(b_0, b_1, s)$ is in Trm_{int}.
5. If vi is a variable of type *int*, b_0, b_1 are terms of type *int* not containing vi and s is an arbitrary term in Trm_{int}, then $bprod\{vi\}(b_0, b_1, s)$ is in Trm_{int}.
6. If vi is a variable of arbitrary type and s a term of type *LocSet*, then $infiniteUnion\{vi\}(s)$ is in Trm_{LocSet}.
7. If vi is a variable of type *int*, b_0, b_1 are terms of type *int* not containing vi and s is an arbitrary term in Trm_{any}, then $seqDef\{vi\}(b_0, b_1, s)$ is in Trm_{Seq}.

It is instructive to observe the role of the quantified variable vi in the following syntax definition:

Definition 2.23 (extends Definition 2.5). If t is one of the terms $bsum\{vi\}(b_0, b_1, s)$, $bprod\{vi\}(b_0, b_1, s)$, $infiniteUnion\{vi\}(s)$, and $seqDef\{vi\}(b_0, b_1, s)$ we have

$$var(t) = var(b_0) \cup var(b_1) \cup var(s) \text{ and } fv(t) = var(t) \setminus \{vi\} \ .$$

We trust that the following remarks will suffice to clarify the semantic meaning of the first two symbols introduced in Definition 2.22. In mathematical notation one would write $\Sigma_{b_0 \leq vi < b_1} s_{vi}$ for $bsum\{vi\}(b_0, b_1, s)$ and $\Pi_{b_0 \leq vi < b_1} s_{vi}$ for $bprod\{vi\}(b_0, b_1, s)$. For the corner case $b_1 \leq b_0$ we stipulate $\Sigma_{b_0 \leq vi < b_1} s_{vi} = 0$ and $\Pi_{b_0 \leq vi < b_1} s_{vi} = 1$. The name *bsum* stands for *bounded sum* to emphasize that infinite sums are not covered. The proof rules for *bsum* and *bprod* are the obvious recursive definitions plus the stipulation for the corner cases which we forgo to reproduce here.

For an integer variable vi the term $infiniteUnion\{vi\}(s)$ would read in mathematical notation $\bigcup_{-\infty < vi < \infty} s$, and analogously for variables vi of type other than integer. The precise semantics is part of Figure 2.11 in Section 2.4.4 below.

The semantics of $seqDef\{vi\}(b_0, b_1, s)$ will be given in Definition 5.2 on page 151. But, it makes an interesting additional example of a binder symbol. The term

$\mathit{seqDef}\{vi\}(b_0, b_1, s)$ is to stand for the finite sequence $\langle s(b_0), s(b_0+1), \ldots, s(b_1 - 1)\rangle$. For $b_1 \leq b_0$ the result is the empty sequence, i.e., $\mathit{seqDef}\{vi\}(b_0, b_1, s) = \langle\rangle$. The proof rules related to *seqDef* are discussed in Chapter 5.

2.3.2 Undefinedness

In KeY all functions are total. There are two ways to interpret a function symbol f in a structure $\mathscr{M}$ at an argument position $\bar{a}$ outside its intended range of definition:

1. The value of the function $\mathit{val}_{\mathscr{M}}(f)$ at position $\bar{a}$ is set to a default within the intended range of f. E.g., $\mathit{bsum}\{vi\}(1,0,s)$ evaluates to 0 (regardless of s).
2. The value of the function $\mathit{val}_{\mathscr{M}}(f)$ at position $\bar{a}$ is set to an arbitrary value b within the intended range of f. For different structures different b are chosen. When we talk about universal validity, i.e., truth in all structures, we assume that for every possible choice of b there is a structure $\mathscr{M}_b$ such that $\mathit{val}_{\mathscr{M}_b}(f)(\bar{a}) = b$. The prime example for this method, called *underspecification*, is division by 0 such that, e.g., $\frac{1}{0}$ is an arbitrary integer.

Another frequently used way to deal with undefinedness is to choose an error element that is different from all defined values of the function. We do not do this. The advantage of underspecification is that no changes to the logic are required. But, one has to know what is happening. In the setting of underspecification we can prove $\exists\, i; (\frac{1}{0} \doteq i)$ for an integer variable i. However, we cannot prove $\frac{1}{0} \doteq \frac{2}{0}$. Also the formula $\mathit{cast}_{\mathit{int}}(c) \doteq 5 \rightarrow c \doteq 5$ is not universally valid. In case c is not of type *int* the underspecified value for $\mathit{cast}_{\mathit{int}}(c)$ could be 5 for $c \neq 5$.

The underspecification method gives no warning when undefined values are used in the verification process. The KeY system offers a well-definedness check for JML contracts, details are described in Section 8.3.3.

2.4 First-Order Logic for Java

As already indicated in the introduction of this chapter, Java first-order logic (JFOL) will be an instantiation of the extended classical first-order logic from Subsection 2.3 tailored towards the verification of Java programs. The precise type hierarchy $\mathscr{T}$ and signature Σ will of course depend on the program and the statements to be proved about it. But we can identify a basic vocabulary that will be useful to have in almost every case. Figure 2.3 shows the type hierarchy $\mathscr{T}_J$ that we require to be at least contained in the type hierarchy $\mathscr{T}$ of any instance of JFOL. The mandatory function and predicate symbols Σ_J are shown in Figure 2.4. Data types are essential for formalizing nontrival program properties. The data types of the integers and the theory of arrays are considered so elementary that they are already included here. More precisely what is covered here are the mathematical integers. There are of course also Java integers types. Those and their relation to the mathematical integers are covered in Section 5.4 on page 161. Also the special data type *LocSet* of sets of memory locations will already be covered here. Why it is essential for the verification of Java programs will become apparent in Chapters 8 and 9. The data type of *Seq* of finite sequences however will extensively be treated later in Section 5.2.

2.4.1 Type Hierarchy and Signature

The mandatory type hierarchy $\mathcal{T}_J$ for JFOL is shown in Figure 2.3. Between *Object* and *Null* the class and interface types from the Java code to be investigated will appear. In the future there might be additional data types at the level immediately below *Any* besides *boolean*, *int*, *LocSet* and *Seq*, e.g., *maps*.

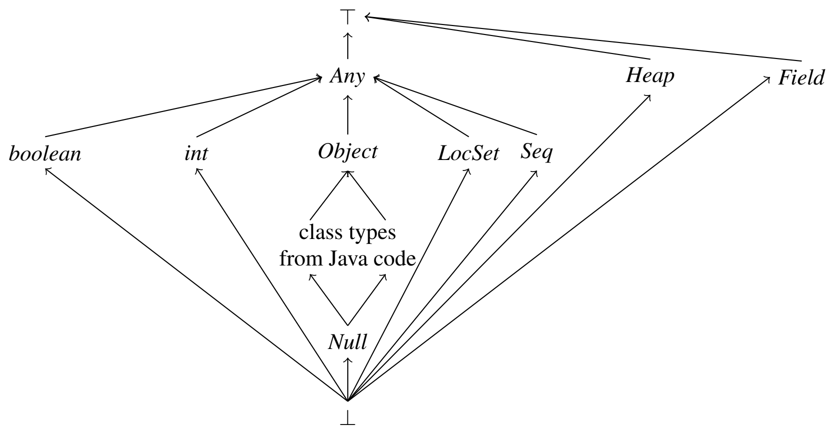

Figure 2.3 The mandatory type hierarchy $\mathcal{T}_J$ of JFOL

The mandatory vocabulary Σ_J of JFOL is shown in Figure 2.4 using the same notation as in Definition 2.2. In the subsections to follow we will first present the axioms that govern these data types one by one and conclude with their model-theoretic semantics in Subsection 2.4.5.

As mentioned above, in the verification of a specific Java program the signature Σ may be a strict superset of Σ_J. To mention just one example: for every model field m of type T contained in the specification of a Java class C an new symbol $f_m : Heap \times C \to T$ is introduced. We will in Definition 9.7 establish the terminology that function symbols with at least one, usually the first, argument of type *Heap* are called *observer function symbols*.

int and *boolean*	all function and predicate symbols for *int*, e.g., $+, *, <, \ldots$
	boolean constants *TRUE*, *FALSE*
Java types	$null : Null$
	$length : Object \rightarrow int$
	$cast_A : Object \rightarrow A$ for any A in $\mathscr{T}$ with $\bot \sqsubset A \sqsubseteq Object$.
	$instance_A : Any \rightarrow boolean$ for any type $A \sqsubseteq Any$
	$exactInstance_A : Any \rightarrow boolean$ for any type $A \sqsubseteq Any$
Field	$created : Field$
	$arr : int \rightarrow Field$
	$f : Field$ for every Java field f
Heap	$select_A : Heap \times Object \times Field \rightarrow A$ for any type $A \sqsubseteq Any$
	$store : Heap \times Object \times Field \times Any \rightarrow Heap$
	$create : Heap \times Object \rightarrow Heap$
	$anon : Heap \times LocSet \times Heap \rightarrow Heap$
	$wellFormed(Heap)$
LocSet	$\varepsilon(Object, Field, LocSet)$
	$empty, allLocs : LocSet$
	$singleton : Object \times Field \rightarrow LocSet$
	$subset(LocSet, LocSet)$
	$disjoint(LocSet, LocSet)$
	$union, intersect, setMinus : LocSet \times LocSet \rightarrow LocSet$
	$allFields : Object \rightarrow LocSet$, $allObjects : Field \rightarrow LocSet$
	$arrayRange : Object \times int \times int \rightarrow LocSet$
	$unusedLocs : Heap \rightarrow LocSet$

Figure 2.4 The mandatory vocabulary Σ_J of JFOL

2.4.2 Axioms for Integers

polySimp_addComm0	$k+i \doteq i+k$	add_zero_right	$i+0 \doteq i$
polySimp_addAssoc	$(i+j)+k \doteq i+(j+k)$	add_sub_elim_right	$i+(-i) \doteq 0$
polySimp_elimOne	$i*1 \doteq i$	mul_distribute_4	$i*(j+k) \doteq (i*j)+(i*k)$
mul_assoc	$(i*j)*k \doteq i*(j*k)$	mul_comm	$j*i \doteq i*j$
less_trans	$i<j \wedge j<k \rightarrow i<k$	less_is_total_heu	$i<j \vee i \doteq j \vee j<i$
less_is_alternative_1	$\neg(i<j \wedge j<i)$	less_literals	$0<1$
add_less	$i<j \rightarrow i+k<j+k$	multiply_inEq	$i<j \wedge 0<k \rightarrow i*k<j*k$

$$\text{int\_induction}\ \frac{\Gamma \Longrightarrow \phi(0), \Delta \quad \Gamma \Longrightarrow \forall n; (0 \leq n \wedge \phi(n) \rightarrow \phi(n+1)), \Delta}{\Gamma \Longrightarrow \forall n; (0 \leq n \rightarrow \phi(n)), \Delta}$$

Figure 2.5 Integer axioms and rules

Figure 2.5 shows the axioms for the integers with $+$, $*$ and $<$. Occasionally we use the additional symbol $\leq$ which is, as usual, defined by $x \leq y \leftrightarrow (x < y \vee x \doteq y)$. The implication multiply_inEq does in truth not occur among the KeY taclets. Instead multiply_inEq0 $i \leq j \wedge 0 \leq k \rightarrow i*k \leq j*k$ is included. But, multiply_inEq can be derived from , multiply_inEq0 although by a rather lengthy proof (65 steps) based on a normal form transformation. The reverse implication is trivially true.

Figure 2.5 also lists in front of each axiom the name of the taclet that implements it. The KeY system not only implements the shown axioms but many useful consequences and defining axioms for further operations such as those related to integer division and the modulo function. How the various integer data types of the Java language are handled in the KeY system is explained in Section 5.4.

Incompleteness

Mathematically the integers $(\mathbb{Z}, +, *, 0, 1, <)$ are a commutative ordered ring satisfying the well-foundedness property: every nonempty subset of the positive integers has a least element. Well-foundedness is a second-order property. It is approximated by the first-order induction schema, which can be interpreted to say that every nonempty definable subset of the positive integers has a least element. The examples known so far of properties of the integers that can be proved in second-order logic but not in its first-order approximation, see e.g. [Kirby and Paris, 1982] are still so arcane that we need not worry about this imperfection.

2.4.3 Axioms for Heap

The state of a Java program is determined by the values of the local variables and the heap. A heap assigns to every pair consisting of an object and a field declared for this object an appropriate value. As a first step to model heaps, we require that a type *Field* be present in JFOL. This type is required to contain the field constant *created* and the fields $arr(i)$ for array access for natural numbers $0 \leq i$. In a specific verification context there will be constants f for every field f occurring in the Java program under verification. There is no assumption, however, that these are the only elements in *Field*; on the contrary, it is completely open which other field elements may occur. This feature is helpful for modular verification: when the contracts for methods in a Java class are verified, they remain true when new fields are added. The data type *Heap* allows us to represent more functions than can possibly occur as heaps in states reachable by a Java program:

1. Values may be stored for arbitrary pairs (o, f) of objects o and fields f regardless of the question if f is declared in the class of o.
2. The value stored for a pair (o, f) need not match the type of f.
3. A heap may assign values for infinitely many objects and fields.

On one hand our heap model allows for heaps that we will never need, on the other hand this generality makes the model simpler. Relaxation 2 in the above list is necessary since JFOL does not use dependent types. To compensate for this shortcoming there has to be a family of observer functions $select_A$, where A ranges over all subtypes of *Any*.

The axiomatization of the data type *Heap*, shown in Figure 2.6, follows the pattern well known from the theory of arrays. The standard reference is [McCarthy, 1962]. There are some changes however. One would expect the following rule $select_A(store(h,o,f,x),o2,f2) \rightsquigarrow$ if $o \doteq o2 \wedge f \doteq f2$ then x else $select_A(h,o2,f2)$. Since the type of x need not be A this easily leads to an ill-typed formula. Thus we need $cast_A(x)$ in place of x. In addition the implicit field *created* gets special treatment. The value of this field should not be manipulated by the *store* function. This explains the additional conjunct $f \dot{\neq} created$ in the axiom. The rule selectOfStore as it is shown below implies $select_A(store(h,o,created,x),o2,f2) \doteq select_A(h,o2,f2)$. Assuming extensionality of heaps this entails $store(h,o,created,x) \doteq h$. The *created* field of a heap can only be changed by the *create* function as detailed by the rule selectOfCreate. This ensures that the value of the *created* field can never be changed from *TRUE* to *FALSE*. Note also, that the object *null* is considered to be created from the start, so it can be excepted from rule selectOfCreate.

selectOfStore $select_A(store(h,o,f,x),o2,f2) \rightsquigarrow$
if $o \doteq o2 \wedge f \doteq f2 \wedge f \dot{\neq} created$ then $cast_A(x)$ else $select_A(h,o2,f2)$

selectOfCreate $select_A(create(h,o),o2,f) \rightsquigarrow$
if $o \doteq o2 \wedge o \dot{\neq} null \wedge f \doteq created$ then $cast_A(TRUE)$ else $select_A(h,o2,f)$

selectOfAnon $select_A(anon(h,s,h'),o,f) \rightsquigarrow$
if$(\varepsilon(o,f,s) \wedge f \dot{\neq} created) \vee \varepsilon(o,f,unusedLocs(h)))$
then $select_A(h',o,f)$ else $select_A(h,o,f)$

with the typing $o,o1,o2 : Object, f,f2 : Field, h,h' : Heap, s : LocSet$

Figure 2.6 Rules for the theory of arrays.

There is another operator, named $anon(h,s,h')$, that returns a *Heap* object. Its meaning is described by the rule selectOfAnon in Figure 2.6: at locations (o,f) in the location set s the resulting heap coincides with h' under the proviso $f \dot{\neq} created$, otherwise it coincides with h. To get an idea when this operator is useful imaging that h is the heap reached at the beginning of a while loop that at most changes locations in a location set s and that h' is a totally unknown heap. Then $anon(h,s,h')$ represents a heap reached after an unknown number of loop iterations. This heap may have more created objects than the initial heap h. Since location sets are not allowed to contain locations with not created objects, see onlyCreatedObjectsAreInLocSets in Figure 2.7, this has to be added as an addition case in rule selectOfAnon. This application scenario also accounts for the name which is short for *anonymize*.

A patiently explained example for the use of *store* and *select* functions can be found in Subsection 15.2.3 on page 526. While SMT solvers can handle expressions containing many occurrences of *store* and *select* quite efficiently, they are a pain in the neck for the human reader. The KeY interface therefore presents those expressions in a pretty printed version, see explanations in Section 16.2 on page 544.

The taclets in Figure 2.6 are called *rewriting taclets*. We use the $\rightsquigarrow$ notation to distinguish them from the other sequent rules as, e.g., in Figures 2.1 and 2.2. A rewriting rule $s \rightsquigarrow t$ is shorthand for a sequent rule $\frac{\Gamma' \Longrightarrow \Delta'}{\Gamma \Longrightarrow \Delta}$ where $\Gamma' \Longrightarrow \Delta'$ arises from $\Gamma \Longrightarrow \Delta$ by replacing one or more occurrences of the term s by t. Rewriting rules will again be discussed in Subsection 4.2.3, page 116.

onlyCreatedObjectsAreReferenced
$wellFormed(h) \rightarrow select_A(h,o,f) \doteq null \vee select_{boolean}(h, select_A(h,o,f), created) \doteq TRUE$

onlyCreatedObjectsAreInLocSets
$wellFormed(h) \wedge \varepsilon(o2, f2, select_{LocSet}(h,o,f)) \rightarrow o2 \doteq null \vee$
$select_{boolen}(h, o2, created) \doteq TRUE$

narrowSelectType
$wellFormed(h) \wedge select_B(h,o,f) \rightarrow select_A(h,o,f)$ where type of f is A and $A \sqsubseteq B$

narrowSelectArrayType
$wellFormed(h) \wedge o \dot{\neq} null \wedge select_B(h,o,arr(i)) \rightarrow select_A(h,o,arr(i))$
where type of o is $A[]$ and $A \sqsubseteq B$

wellFormedStoreObject
$wellFormed(h) \wedge (x \doteq null \vee (select_{boolean}(h,x,created) \doteq TRUE \wedge instance_A(x) \doteq TRUE))$
$\rightarrow wellFormed(store(h,o,f,x))$ where type of f is A

wellFormedStoreArray
$wellFormed(h) \wedge (x \doteq null \vee (select_{boolean}(h,x,created) \doteq TRUE \wedge arrayStoreValid(o,x)))$
$\rightarrow wellFormed(store(h,o,arr(idx),x)))$

wellFormedStoreLocSet
$wellFormed(h) \wedge \forall ov; \forall fv; (\varepsilon(ov,fv,y) \rightarrow ov \doteq null \vee select_{boolean}(h,ov,created) \doteq TRUE)$
$\rightarrow wellFormed(store(h,o,f,y))$ where type of f is A and $LocSet \sqsubseteq A$

wellFormedStorePrimitive
$wellFormed(h) \rightarrow wellFormed(store(h,o,f,x))$
provided f is a field of type A, x is of type B, and $B \sqsubseteq A, B \not\sqsubseteq Object, B \not\sqsubseteq LocSet$

wellFormedStorePrimitiveArray
$wellFormed(h) \rightarrow wellFormed(store(h,o,arr(idx),x))$
provided o is of sort A, x is of sort $B, B \not\sqsubseteq Object, B \not\sqsubseteq LocSet, B \sqsubseteq A$

wellFormedCreate
$wellFormed(h) \rightarrow wellFormed(create(h,o))$

wellFormedAnon
$wellFormed(h) \wedge wellFormed(h2) \rightarrow wellFormed(anon(h,y,h2))$

In the above formulas the following implicitly universally quantified variables are used: $h, h2 : Heap$, $o, x : Object$, $f : Field$, $i : int$, $y : LocSet$

Figure 2.7 Rules for the predicate *wellFormed*

Our concept of heap is an overgeneralization. Most of the time this does no harm. But, there are situations where it is useful to establish and depend on certain well-formedness conditions. The predicate $wellFormed(heap)$ has been included in the vocabulary for this purpose. No effort is made to make the $wellFormed(h)$ predicate so strong that it only is true of heaps h that can actually occur in Java programs. The axioms in Figure 2.7 were chosen on a pragmatic basis. There is e.g., no axiom that guarantees for a created object o of type A with $select(h,o,f)$ defined that the field f is declared in class A.

The first four axioms in Figure 2.7 formalize properties of well-formed heaps while the rest cover situations starting out with a well-formed heap, manipulate it and end up again with a well-formed heap. The formulas are quite self-explanatory. Reading though them you will encounter the auxiliary predicate `arrayStoreValid`: `arrayStoreValid(o,x)` is true if o is an array object of exact type $A[]$ and x is of type A.

The meaning of the functions symbols $instance_A(x)$, $exactInstance_A(x)$, $cast_A(x)$, and $length(x)$ is given by the axioms in Figure 2.8. This time we present the axioms in mathematical notation for conciseness. The axiom scheme, (Ax-I) and (Ax-C) show that adding $instance_A$ and $cast_A$ does not increase the expressive power. These functions can be defined already in the basic logic plus underspecification. The formulas (Ax-E_1) and (Ax-E_2) completely axiomatize the $exactInstance_A$ functions, see Lemma 2.24 on page 47. The function $length$ is only required to be not negative. Axioms (Ax-E_1), (Ax-E_2), and (Ax-L) are directly formalized in the KeY system as

$$\begin{array}{ll}
\forall Object\, x; (instance_A(x) \doteq TRUE \leftrightarrow \exists y; (y \doteq x)) \text{ with } y : A & \text{(Ax-I)} \\
\forall Object\, x; (exactInstance_A(x) \doteq TRUE \rightarrow instance_A(x) \doteq TRUE) & \text{(Ax-E}_1\text{)} \\
\forall Object\, x; (exactInstance_A(x) \doteq TRUE \rightarrow instance_B(x) \doteq FALSE) \text{ with } A \not\sqsubseteq B & \text{(Ax-E}_2\text{)} \\
\forall Object\, x; (instance_A(x) \doteq TRUE \rightarrow cast_A(x) \doteq x) & \text{(Ax-C)} \\
\forall Object\, x; (length(x) \geq 0) & \text{(Ax-L)}
\end{array}$$

Figure 2.8 Axioms for functions related to Java types

taclets instance_known_dynamic_type, exact_instance_known_dynamic_type and arrayLengthNotNegative. The other two axioms families have no direct taclet counterpart. But, they can easily be derived.

2.4.4 Axioms for Location Sets

The data type $LocSet$ is a very special case of the set type in that only sets of heap locations are considered, i.e., sets of pairs (o, f) with o an object and f a field. This immediately guarantees that the is-element-of relation ε is well-foundedfor $LocSet$. Problematic formulas such as $a \varepsilon a$ are already syntactically impossible.

The rules for the data type *LocSet* are displayed in Figure 2.9. The only constraint on the membership relation ε is formulated in rule equalityToElementOf. One could view this rule as a definition of equality for location sets. But, since equality is a built in relation in the basic logic it is in fact a constraint on ε. All other rules in this figure are definitions of the additional symbols of the data type, such as, e.g., *allLocs*, *union*, *intersect*, and $\mathit{infiniteUnion}\{av\}(s1)$.

elementOfEmpty	$\varepsilon(o1, f1, \mathit{empty})$	$\rightsquigarrow \mathit{FALSE}$
elementOfAllLocs	$\varepsilon(o1, f1, \mathit{allLocs})$	$\rightsquigarrow \mathit{TRUE}$
equalityToElementOf	$s1 \doteq s2$	$\rightsquigarrow \forall o; \forall f; (\varepsilon(o, f, s1) \leftrightarrow \varepsilon(o, f, s2))$
elementOfSingleton	$\varepsilon(o1, f1, \mathit{singleton}(o2, f2)$	$\rightsquigarrow o1 \doteq o2 \land f1 \doteq f2$
elementOfUnion	$\varepsilon(o1, f1, \mathit{union}(t1, t2))$	$\rightsquigarrow \varepsilon(o1, f1, t1) \lor \varepsilon(o1, f1, t2)$
subsetToElementOf	$\mathit{subset}(t1, t2)$	$\rightsquigarrow \forall o; \forall f; (\varepsilon(o, f, t1) \rightarrow \varepsilon(o, f, t2))$
elementOfIntersect	$\varepsilon(o1, f1, \mathit{intersect}(t1, t2))$	$\rightsquigarrow \varepsilon(o1, f1, t1) \land \varepsilon(o1, f1, t2)$
elementOfAllFields	$\varepsilon(o1, f1, \mathit{allFields}(o2)$	$\rightsquigarrow o1 \doteq o2$
elementOfSetMinus	$\varepsilon(o1, f1, \mathit{setMinus}(t1, t2))$	$\rightsquigarrow \varepsilon(o1, f1, t1) \land \neg\varepsilon(o1, f1, t2)$
elementOfAllObjects	$\varepsilon(o1, f1, \mathit{allObjects}(f2)$	$\rightsquigarrow f1 \doteq f2$
elementOfInfiniteUnion	$\varepsilon(o1, f1, \mathit{infiniteUnion}\{av\}(s1))$	$\rightsquigarrow \exists av; \varepsilon(o1, f1, s1)$

with the typing $o, o1, 02 : \mathit{Object}, f, f1 : \mathit{Field}, s1, s2, t1, t2 : \mathit{LocSet}$, *av* of arbitrary type.

Figure 2.9 Rules for data type *LocSet*

2.4.5 Semantics

As already remarked at the start of Subsection 2.2.3, a formal semantics opens up the possibility for rigorous soundness and relative completeness proofs. Here we extend and adapt the semantics provided there to cover the additional syntax introduced for JFOL (see Section 2.4.1).

We take the liberty to use an alternative notion for the interpretation of terms. While we used $\mathrm{val}_{\mathscr{M},\beta}(t)$ in Section 2.2.3 to emphasize also visually that we are concerned with evaluation, we will write $t^{\mathscr{M},\beta}$ for brevity here.

The definition of a FOL structure $\mathscr{M}$ for a given signature in Subsection 2.2.3 was deliberately formulated as general as possible, to underline the universal nature of logic. The focus in this subsection is on semantic structures tailored towards the verification of Java programs. To emphasize this perspective we call these structures JFOL structures.

A decisive difference to the semantics from Section 2.2.3 is that now the interpretation of some symbols, types, functions, predicates, is constrained. Some functions are completely fixed, e.g., addition and multiplication of integers. Others are almost fixed, e.g., integer division n/m that is fixed except for $n/0$ which may have different interpretations in different structures. Other symbols are only loosely constrained, e.g., *length* is only required to be nonnegative.

The semantic constraints on the JFOL type symbols are shown in Figure 2.10. The restriction on the semantics of the subtypes of *Object* is that their domains contain for every $n \in \mathbb{N}$ infinitely many elements o with $length^{\mathcal{M}}(o) = n$. The reason for this is the way object creation is modeled. When an array object is created an element o in the corresponding type domain is provided whose *created* field has value *FALSE*. The created field is then set to *TRUE*. Since the function *length* is independent of the heap it cannot be changed in the creation process. So, the element picked must already have the desired length. This topic will be covered in detail in Subsection 3.6.6. The semantics of *Seq* will be given in Chapter 5.

- $D^{int} = \mathbb{Z}$,
- $D^{boolean} = \{tt, ff\}$,
- $D^{ObjectType}$ is an infinite set of elements for every *ObjectType* with $Null \sqsubset ObjectType \sqsubseteq Object$, subject to the restriction that for every positive integer n there are infinitely many elements o in $D^{ObjectType}$ with $length^{\mathcal{M}}(o) = n$.
- $D^{Null} = \{null\}$,
- D^{Heap} = the set of all functions $h : D^{Object} \times D^{Field} \to D^{Any}$,
- D^{LocSet} = the set of all subsets of $\{(o, f) \mid o \in D^{Object}$ and $f \in D^{Field}\}$,
- D^{Field} is an infinite set.

Figure 2.10 Semantics on type domains

Constant Domain

Let T be a theory, that does not have finite models. By definition $T \vdash \phi$ iff $\mathcal{M} \models \phi$ for all models $\mathcal{M}$ of T. The Löwenheim-Skolem Theorem, which by the way follows easily from the usual completeness proofs, guarantees that $T \vdash \phi$ iff $\mathcal{M} \models \phi$ for all countably infinite models $\mathcal{M}$ of T. Let S be an arbitrary countably infinite set, then we have further $T \vdash \phi$ iff $\mathcal{M} \models \phi$ for all models $\mathcal{M}$ of T such that the universe of $\mathcal{M}$ is S. To see this assume there is a countably infinite model $\mathcal{N}$ of T with universe N such that $\mathcal{N} \models \neg\phi$. For cardinality reasons there is a bijection b from N onto S. So far, S is just a set. It is straightforward to define a structure $\mathcal{M}$ with universe S such that b is an isomorphism from $\mathcal{N}$ onto $\mathcal{M}$. This entails the contradiction $\mathcal{M} \models \neg\phi$.

The interpretation of all the JFOL function and predicate symbols listed in Figure 2.4 is at least partly fixed. All JFOL structures $\mathcal{M} = (M, \delta, I)$ are required to satisfy the constraints put forth in Figure 2.11.

Some of these constraints are worth an explanation. The semantics of the *store* function, as stated above, is such that it cannot change the implicit field *created*. Also there is no requirement that the type of the value x should match with the type of the

1. $\mathit{TRUE}^{\mathcal{M}} = tt$ and $\mathit{FALSE}^{\mathcal{M}} = \mathit{ff}$
2. $\mathit{select}_A^{\mathcal{M}}(h,o,f) = \mathit{cast}_A^{\mathcal{M}}(h(o,f))$
3. $\mathit{store}^{\mathcal{M}}(h,o,f,x) = h^*$, where the function h^* is defined by
$$h^*(o',f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq \mathit{created}^{\mathcal{M}} \\ h(o',f') & \text{otherwise} \end{cases}$$
4. $\mathit{create}^{\mathcal{M}}(h,o) = h^*$, where the function h^* is defined by
$$h^*(o',f) = \begin{cases} tt & \text{if } o' = o, o \neq \mathit{null} \text{ and } f = \mathit{created}^{\mathcal{M}} \\ h(o',f) & \text{otherwise} \end{cases}$$
5. $\mathit{arr}^{\mathcal{M}}$ is an injective function from $\mathbb{Z}$ into $\mathit{Field}^{\mathcal{M}}$
6. $\mathit{created}^{\mathcal{M}}$ and $f^{\mathcal{M}}$ for each Java field f are elements of $\mathit{Field}^{\mathcal{M}}$, which are pairwise different and also not in the range of $\mathit{arr}^{\mathcal{M}}$.
7. $\mathit{null}^{\mathcal{M}} = \mathit{null}$
8. $\mathit{cast}_A^{\mathcal{M}}(o) = \begin{cases} o & \text{if } o \in A^{\mathcal{M}} \\ \text{arbitrary element in } A^{\mathcal{M}} & \text{otherwise} \end{cases}$
9. $\mathit{instance}_A(o)^{\mathcal{M}} = tt \Leftrightarrow o \in A^{\mathcal{M}} \Leftrightarrow \delta(o) \sqsubseteq A$
10. $\mathit{exactInstance}_A^{\mathcal{M}} = tt \Leftrightarrow \delta(o) = A$
11. $\mathit{length}^{\mathcal{M}}(o) \in \mathbb{N}$
12. $\langle o,f,s\rangle \in \varepsilon^{\mathcal{M}}$ iff $(o,f) \in s$
13. $\mathit{empty}^{\mathcal{M}} = \emptyset$
14. $\mathit{allLocs}^{\mathcal{M}} = \mathit{Object}^{\mathcal{M}} \times \mathit{Field}^{\mathcal{M}}$
15. $\mathit{singleton}^{\mathcal{M}}(o,f) = \{(o,f)\}$
16. $\langle s_1,s_2\rangle \in \mathit{subset}^{\mathcal{M}}$ iff $s_1 \subseteq s_2$
17. $\langle s_1,s_2\rangle \in \mathit{disjoint}^{\mathcal{M}}$ iff $s_1 \cap s_2 = \emptyset$
18. $\mathit{union}^{\mathcal{M}}(s_1,s_2) = s_1 \cup s_2$
19. $\mathit{infiniteUnion}\{av\}(s)^{\mathcal{M}} = \{(a \in D^T \mid s^{\mathcal{M}}[a/av]\}$ with T type of av
20. $\mathit{intersect}^{\mathcal{M}}(s_1,s_2) = s_1 \cap s_2$
21. $\mathit{setMinus}^{\mathcal{M}}(s_1,s_2) = s_1 \setminus s_2$
22. $\mathit{allFields}^{\mathcal{M}}(o) = \{(o,f) \mid f \in \mathit{Field}^{\mathcal{M}}\}$
23. $\mathit{allObjects}^{\mathcal{M}}(f) = \{(o,f) \mid o \in \mathit{Object}^{\mathcal{M}}\}$
24. $\mathit{arrayRange}^{\mathcal{M}}(o,i,j) = \{(o,\mathit{arr}^{\mathcal{M}}(x) \mid x \in \mathbb{Z}, i \leq x \leq j\}$
25. $\mathit{unusedLocs}^{\mathcal{M}}(h) = \{(o,f) \mid o \in \mathit{Object}^{\mathcal{M}}, f \in \mathit{Field}^{\mathcal{M}}, o \neq \mathit{null}, h(o,\mathit{created}^{\mathcal{M}}) = \mathit{false}\}$
26. $\mathit{anon}^{\mathcal{M}}(h_1,s,h_2) = h^*$, where the function h^* is defined by:
$$h^*(o,f) = \begin{cases} h_2(o,f) \text{ if } (o,f) \in s \text{ and } f \neq \mathit{created}^{\mathcal{M}}, \text{ or} \\ \qquad\qquad (o,f) \in \mathit{unusedLocs}^{\mathcal{M}}(h_1) \\ h_1(o,f) \text{ otherwise} \end{cases}$$

Figure 2.11 Semantics for the mandatory JFOL vocabulary (see Figure 2.4)

field f. This liberality necessitates the use of the cast_A functions in the semantics of select_A.

It is worth pointing out that the length function is defined for all elements in D^{Object}, not only for elements in D^{OT} where OT is an array type.

Since the semantics of the wellFormed predicate is a bit more involved we put it separately in Figure 2.12

The integer operations are defined as usual with the following versions of integer division and the modulo function:

$$
h \in \mathit{wellFormed}^{\mathcal{M}} \text{ iff } \begin{array}{ll}
(a) & \text{if } h(o,f) \in D^{Object} \text{ then } h(o,f) = \mathit{null} \text{ or } h(h(o,f), \mathit{created}^{\mathcal{M}}) = \mathit{tt} \\
(b) & \text{if } h(o,f) \in D^{LocSet} \text{ then } nh(o,f) \cap \mathit{unusedLocs}^{\mathcal{M}}(h) = \emptyset \\
(c) & \text{if } \delta(o) = T[] \text{ then } \delta(h(o, \mathit{arr}^{\mathcal{M}}(i))) \sqsubseteq T \text{ for all } 0 \leq i < \mathit{length}^{\mathcal{M}}(o) \\
(d) & \text{there are only finitely many } o \in D^{Object} \text{ for which } h(o, \mathit{created}^{\mathcal{M}}) = \mathit{tt}
\end{array}
$$

Figure 2.12 Semantics for the predicate *wellFormed*

$$
n /^{\mathcal{M}} m = \begin{cases}
\begin{array}{l}\text{the uniquely defined } k \text{ such that} \\ |m| * |k| \leq |n| \text{ and } |m| * (|k|+1) > |n| \text{ and} \\ k \geq 0 \text{ if } m, n \text{ are both positive or both negative and} \\ k \leq 0 \text{ otherwise}\end{array} & \text{if } m \neq 0 \\
\\
\text{unspecified} & \text{otherwise}
\end{cases}
$$

Thus integer division is a total function with arbitrary values for $x /^{\mathcal{M}} 0$. Division is an example of a partially fixed function. The interpretation of $/$ in a JFOL structure $\mathcal{M}$ is fixed except for the values $x /^{\mathcal{M}} 0$. These may be different in different JFOL structures. The modulo function is defined by

$$
\text{mod}(n,d) = n - (n/d) * d
$$

Note, that this implies $\text{mod}(n,0) = n$ as $/$ is – due to using underspecification – a total function.

Lemma 2.24. *The axioms in Figure 2.8 are sound and complete with respect to the given semantics.*

For a proof see [Schmitt and Ulbrich, 2015].

http://www.springer.com/978-3-319-49811-9

Deductive Software Verification – The KeY Book
From Theory to Practice
Ahrendt, W.; Beckert, B.; Bubel, R.; Hähnle, R.; Schmitt, P.H.; Ulbrich, M. (Eds.)
2016, XXXII, 702 p. 110 illus., Softcover
ISBN: 978-3-319-49811-9