Chapter 2
First-order Logic

Peter H. Schmitt

chap:fol

Abstract

The ultimate goal of first-order logic in the context of this book, and this applies to a great extend also to Computer Science in general, is the formalisation of and reasoning with natural language specifications of systems and programs. This chapter provides the logical foundations for doing so in three steps. In Sect. 2.1 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 taylored 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 Sect. 2.2 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 Sect. 2.3 the extended basic logic will be instantiated to JFOL taylored 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 scence in Chapter 3

2.1 Basic First-Order Logic

sec02:BasicFOL

2.1.1 Syntax

subsec02:BasicFOLSyntax

def:typehier

Definition 2.1. A *type hierarchy* $\mathscr{T} = (\text{TSym}, \sqsubseteq)$ consists of

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

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 $B \sqsubseteq A$.

def:Signature **Definition 2.2.** A *signature* $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym})$ for a given type hierarchy $\mathscr{T}$ is made up of

1. a set FSym of typed function symbols,
   by $f : A_1, \ldots, A_n \to A$ 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,
02:item:SignaturePSym 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 $c : \to A$ is called a constant symbol of type A. A 0-ary predicate symbol $p : ()$ is called a 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.

def:TermA **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, $A \neq \bot$, is inductively defined by

1. $v \in \mathrm{Trm}_A$ for each variable symbol $v : A \in \mathrm{VSym}$ of type A.
item:termComposition 2. $f(t_1, \ldots, t_n) \in \mathrm{Trm}_A$ for each $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}$ and terms $t_i \in \mathrm{Trm}_{B_i}$ with $B_i \sqsubseteq A_i$ for all $1 \leq i \leq n$.
item:condTerm 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 $\sigma(t) = A$.

Note, that item (2) in Definition 3 entails $c \in \mathrm{Trm}_A$ for each constant symbol $c : \to A \in \mathrm{FSym}$. 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 σ.

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.
02MereConvenience

def:FolFml **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. $!\phi$, $\phi \,\&\, \psi$, $\phi \mid \psi$, $\phi \mathrel{->} \psi$, $\phi \leftrightarrow \psi$ 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 $v : A \in \mathrm{VSym}$.

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*.

def:FreeBoundVars

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

$var(v) =$	$\{v\}$	$fv(v) =$	$\{v\}$	for $v \in \mathrm{VSym}$
$var(t) =$	$\bigcup_{i=1}^n var(t_i)$	$fv(t) =$	$\bigcup_{1=i}^n fv(t_i)$	for $t = f(t_1,\ldots,t_n)$
$var(t) =$	$var(\phi) \cup var(t_1) \cup var(t_2)$	$fv(t) =$	$fv(\phi) \cup fv(t_1) \cup fv(t_2)$	for $t =$ (if ϕ then t_1 else t_2)
$var(\phi) =$	$\bigcup_{i=1}^n var(t_i)$	$fv(\phi) =$	$\bigcup_{i=1}^n fv(t_i)$	for $\phi = p(t_1,\ldots,t_n)$
$var(!\,\phi) =$	$var(\phi)$	$fv(!\,\phi) =$	$fv(\phi)$	
$var(\phi) =$	$var(\phi_1) \cup var(\phi_2)$	$fv(\phi) =$	$fv(\phi_1) \cup fv(\phi_2)$	for $\phi = \phi_1 \circ \phi_2$ where $\circ$ is any binary Boolean operation
$var(Q\ v.\phi) = var(\phi)$		$fv(Q\ v.\phi) = var(\phi) \setminus \{v\}$		where $Q \in \{\forall, \exists\}$

A term without free variables is called a *ground term*.

It is an obvious consequence of this definition that every occurence 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.

def:Substitution

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.

def:applySubst **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. $\tau(t_1 \doteq t_k) = \tau(_1) \doteq \tau(t_k)$
7. $\tau(!\,\phi) = !\,\tau(\phi)$
8. $\tau(\phi_1 \circ \phi_2) = \tau(\phi_1) \circ \tau(\phi_2)$ for propositional operators $\circ \in \{\&, |, ->, \leftrightarrow\}$
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 $x \in \mathrm{dom}(\tau_v)$.

item:def:applySubstQ

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 Def. 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 non-ground substitutions to formulas. Care has to be taken in that case to avoid *clashes*, see Example 2.8 below. We will only need ground substiutions later on, so we sidestep this difficulty.

ex:Subst *Example 2.8.* For the sake of this example we assume that there is a type symbol $\mathit{int} \in \mathrm{TSym}$, function symbols $+ : \mathit{int},\mathit{int} \to \mathit{int}$, $* : \mathit{int},\mathit{int} \to \mathit{int}$, $- : \mathit{int} \to \mathit{int}$, $\mathit{exp} : \mathit{int},\mathit{int} \to \mathit{int}$ 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 further more $x : \mathit{int}$, $y : \mathit{int}$ 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

$\phi_1 = \forall x; ((x+y)^2 \doteq x^2 + 2xy + y^2)$	$\tau_1(\phi_1) = \forall x; ((x+1)^2 \doteq x^2 + 2*x*1 + 1^2)$
$\phi_2 = (x+y)^2 \doteq x^2 + 2xy + y^2$	$\tau_1(\phi_2) = (0+1)^2 \doteq 0^2 + 2*0*1 + 1^2$
$\phi_3 = \exists x; (x > y)$	$\tau_1(\phi_3) = \exists x; (x > 1)$

Application of the non-ground substitution $\tau_2 = [y/x]$ on ϕ_3 leads to $\exists x; ((x > x)$. While $\exists x; (x > y)$ is true for all 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 occurence of the term t_1. How should be 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 refered 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 Fig. 2.1 and 2.2.

2.1.2 Calculus

subsec02:BasicFOLCalculus

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 Chap. 4. Chap. 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 premises or the antecedent of the sequent; the formulas $\psi_1,\ldots,\psi_m$ on the right are the conclusions or the succedent. The intended meaning of a sequent is that the premises together imply at least one conclusion. In other words, a sequent $\phi_1,\ldots,\phi_n \Longrightarrow \psi_1,\ldots,\psi_m$ is valid iff the formula $\bigwedge_{1=i}^{n}\phi_i \rightarrow \bigvee_{1=j}^{m}\psi_j$ is valid.

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]. Note, that the rules 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 schematic variables in premise and conclusion by the corresponding entities: sets of formulas, formulas, etc.

Gallier87

defi:closingRules

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

defi:proofTree

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 premise is $\Gamma \Longrightarrow \Delta$ and the child node, or children nodes of n are labeled with the conclusion or conclusions of the rule instance.

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

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

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

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

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow [x/c](\phi),\Delta}{\Gamma \Longrightarrow \forall x;\phi,\Delta} \quad \text{with } c : \to A \text{ a new constant, if } x{:}A \qquad \mathsf{allLeft}\ \frac{\Gamma,\forall x;\phi,[x/t](\phi) \Longrightarrow \Delta}{\Gamma,\forall x;\phi \Longrightarrow \Delta} \quad \text{with } t \in \mathrm{Trm}_{A'} \text{ ground, } A' \sqsubseteq A, \text{ if } x{:}A$$

$$\mathsf{exLeft}\ \frac{\Gamma,[x/c](\phi) \Longrightarrow \Delta}{\Gamma,\exists x;\phi \Longrightarrow \Delta} \quad \text{with } c : \to A \text{ a new constant, if } x{:}A \qquad \mathsf{exRight}\ \frac{\Gamma \Longrightarrow \exists x;\phi,[x/t](\phi),\Delta}{\Gamma \Longrightarrow \exists x;\phi,\Delta} \quad \text{with } t \in \mathrm{Trm}_{A'} \text{ ground, } A' \sqsubseteq A, \text{ if } x{:}A$$

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

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

Fig. 2.1 Classical first-order rules

fig:folrules

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 labled by $\Gamma \Longrightarrow \Delta$.

As a first simple example, we will derive the sequent $\Longrightarrow p\ \&\ q \texttt{ -> } q\ \&\ p$. The same formula is also used in the explantation of the KeY prover in Chap. 15. As its antecedent is empty this sequent says that the propositional formula $p\ \&\ q \texttt{ -> } q\ \&\ p$ is a tautology. Application of the rule impRight reduces our proof goal to $p\ \&\ q \Longrightarrow q\ \&\ p$ and application of andLeft further to $p,q \Longrightarrow q\ \&\ p$. 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

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{p,q \Longrightarrow q} \qquad \cfrac{*}{p,q \Longrightarrow p}}{p,q \Longrightarrow q\ \&\ p}}{p\ \&\ q \Longrightarrow q\ \&\ p}}{\Longrightarrow p\ \&\ q \texttt{ -> } q\ \&\ p}$$

Let us look at an example deriviation 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, non-branching, proof tree for the formula $\exists v;\forall w;p(v,w) \mathrel{->} \forall w;\exists v;p(v,w)$:

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

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 $c :\rightarrow A$ and $d :\rightarrow A$. 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.

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

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

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

$$\textsf{eqDynamicSort}\ \frac{\Gamma,t_1 \doteq t_2,\exists x;(x \doteq t_1 \wedge x \doteq t_2) \Longrightarrow \Delta}{\Gamma,t_1 \doteq t_2 \Longrightarrow \Delta}$$

if $\sigma(t_1)$ and $\sigma(t_2)$ are incomparable,
the sort C of x is new and satisfies $C \sqsubset \sigma(t_1)$ and $C \sqsubset \sigma(t_2)$

Fig. 2.2 Equality rules

fig:eqrules

The rules involving equality are shown in Fig. 2.2. The rules eqLeft and eqRight formalize the intuitve 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 additional constraint $\sigma(t_2) \sqsubset \sigma(t_1)$ is necessary. We give an exampel that the rule eqLeftWrong without this restriction is unsound. Consider two types $A \neq B$ with $B \sqsubseteq A$, two constant symbols $a : \rightarrow A$ and $b : \rightarrow B$, and a unary predicate $p : B$. Applying eqLeftWrong 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. The rule eqDynamicSort is essential to ensure completeness of the calculus. It parallels the exLeft and allRight rules in that new symbols are introduced.

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

$$
\begin{array}{ll}
7 & * \\
6 & (a+b)+(c+d) \doteq a+(b+(c+d)), \forall x,y,z;((x+y)+z \doteq x+(y+z)) \\
 & (b+c)+d \doteq b+(c+d) \Longrightarrow \\
 & \qquad\qquad (a+b)+(c+d) \doteq a+(b+(c+d)) \\
5 & (a+b)+(c+d) \doteq a+((b+c)+d), \forall x,y,z;((x+y)+z \doteq x+(y+z)) \\
 & (b+c)+d \doteq b+(c+d) \Longrightarrow \\
 & \qquad\qquad (a+b)+(c+d) \doteq a+(b+(c+d)) \\
4 & (a+b)+(c+d) \doteq a+((b+c)+d), \forall x,y,z;((x+y)+z \doteq x+(y+z)) \Longrightarrow \\
 & \qquad\qquad (a+b)+(c+d) \doteq a+(b+(c+d)) \\
3 & \forall x,y,z;((x+y)+z \doteq x+(y+z)) \Longrightarrow (a+b)+(c+d) \doteq a+((b+c)+d) \\
2 & \forall x,y,z;((x+y)+z \doteq x+(y+z)) \Longrightarrow \\
 & \qquad\qquad \forall x,y,z,u;((x+y)+(z+u) \doteq x+((y+z)+u)) \\
1 & \Longrightarrow \forall x,y,z;((x+y)+z \doteq x+(y+z)) \; -> \\
 & \qquad\qquad \forall x,y,z,u;((x+y)+(z+u) \doteq x+((y+z)+u))
\end{array}
$$

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. Line 4 in turn is arrived at by an application of allLeft with the substitution $[x/a, y/b, z/(b+d)]$. 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 lefthand side of the sequent the term $(b+c)+d$ in $(a+b)+(c+d) \doteq a+(b+(c+d))$ by the righthand 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 aritmetic or the theory of arrays.

2.1.3 Semantics

subsec02:BasicFOLSemantics

So far we trusted that the logical rules contained in Figs. 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 proof completeness.

def:FolUniverse **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 $\delta : D \to \mathrm{TSym} \setminus \{\bot\}$ such that for every $A \in \mathrm{TSym}$ the set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ is not emtpy.

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 $A, B \in \mathrm{TSym} \setminus \{\bot\}$ there is an element $o \in D^A \cap D^B$ only if there exists $C \in \mathrm{TSym}$, $C \neq \bot$ with $C \sqsubseteq A$ and $C \sqsubseteq B$.

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

1. $D^\bot = \emptyset$, $D^\top = D$,
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.

def:FolStructure **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

item:FSymInterp 1. $I(f)$ is a function from $D^{A_1} \times \cdots \times D^{A_n}$ into D^A for $f : A_1, \ldots, A_n \to A$ in FSym,
item:PSymInterp 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 emtpy 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.

def:VarAssignment **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 assigment 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.

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

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

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

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

For a 0-ary? 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 Def. 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.

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

lem:THExt **Lemma 2.18.** *Let $\mathscr{T}_2 = (\mathrm{ATSym}_2, \sqsubseteq_2)$ be an extension of $\mathscr{T}_1 = (\mathrm{ATSym}_1, \sqsubseteq_1)$ 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 $A, B \in \mathrm{TSym}_1$, $C \in \mathrm{TSym}_2 \setminus \mathrm{TSym}_1$, $D \in \mathrm{TSym}_2$

item:THExt-1
item:THExt-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. □

def:LogicConsequence

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 $\Phi \vdash \phi$ 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 $\emptyset \vdash \phi$.
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:

- $\Phi \vdash \phi$ iff the formula $\bigwedge \Phi \to \phi$ 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 the concept $\Phi \vdash_{\mathscr{T},\Sigma} \phi$ 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 $\vdash_{\mathscr{T}_1} \phi_1$. 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, $\vdash_{\mathscr{T}_2} \phi_1$ 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 $\nvdash \phi_1$.

thm:FOLcompleteness

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. Assume that for every type* $A \in \mathscr{T}$ *there is a constant symbol of type* A' *with* $A' \sqsubseteq A$.

Then:

ϕ *is universally valid iff there is a closed proof tree for the sequent* $\Longrightarrow \phi$.

For the untyped calculus a proof of the 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 syntactic constraint placed on Theorem 2.20 the calculus implemented in the KeY systems takes a slightly different but equivalent approach: instead of requiring the existence of sufficient constants is allows to derive via the taclet **ex_unused** for every $A \in \mathscr{T}$ the formula $\exists x(x \doteq x)$ with x a variable fo type A.

An important step in the proof of Theorem 2.20 is the reduction of the claim of the theorem to the statement that all rules are sound and complete in the following sense

def:soundRules **Definition 2.21.** A rule

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

of the sequent calculus is *sound and complete* if: $\Gamma \Longrightarrow \Delta$ is universally valid iff both $\Gamma_1 \Longrightarrow \Delta_1$ and $\Gamma_2 \Longrightarrow \Delta_2$ are universally valid.

The implication from left to right is the completeness part, the reverse implication is the soundness part. For non-branching rules and rules with side conditions the obvious modifications have to be made.

2.2 Extended First-Order Logic

sec02:ExtFOL

In this section we extend the Basic First-Order Logic from Section 2.1. First we turn our attention in Subsection 2.2.1 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 2.2.2. In essence this amounts to replacing a partial function by all its extensions to total functions.

2.2.1 Variable Binders

subsec02:VarBinders

This subsection assumes that the types *int* of mathematical integers, *LocSet* of sets of loctions, and *Seq* the type of finite sequences are present in TSym. For the logic JFOL to be presented in Subsection 2.3 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.

ulbrich13 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. ulbrich13

Definition 2.22 (extends Def. 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}.

item:infUnionDefbinder

6. If *vi* is a variable of arbitrary type and *s* a term of type *LocSet*, then $infiniteUnion\{vi\}(s)$ is in Trm_{LocSet}.

item:seqDefbinder

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}.

def:BinderSymbols

In mathematical notation one would write $\Sigma_{b_0\le vi<b_1} s_{vi}$ for $bsum\{vi\}(b_0,b_1,s)$ and $\Pi_{b_0\le vi<b_1} s_{vi}$ for $bprod\{vi\}(b_0,b_1,s)$. For the corner case $b_1 \le b_0$ we stipulate $\Sigma_{b_0\le vi<b_1} s_{vi} = 0$ and $\Pi_{b_0\le vi<b_1} s_{vi} = 1$. The name *bsum* stands for *bounded sum* to emphasize that infinite sums are not covered.

Likewise $infiniteUnion\{i\}(s)$ for an integer variable *i* would read in mathematical notation $\bigcup_{-\infty<vi<\infty} s$, and analogously for variables *vi* of type other than integer. For *o* a variable of type *Object* and *f* a constant of type *Field* the term $infiniteUnion\{o\}(singleton(o,f))$ denotes the same set as $allObjects(f)$.

With item 7 we are getting a bit ahead of ourselves since the data type *Seq* of finite sequences will be covered only later in Chapter 5. The semantics of $seqDef\{vi\}(b_0,b_1,s)$ will be given in Definition 5.2 on page 121. But, it makes an interesting additional example of a binder symbol. The term $seqDef\{vi\}(b_0,b_1,s)$ is to stand for the finite sequence $\langle s(b_0), s(b_0+1), \ldots, s(b_1-1)\rangle$. For $b_1 \le b_0$ we stipulate that the result is the empty sequence, i.e., $seqDef\{vi\}(b_0,b_1,s) = \langle\rangle$.

def:FreeBoundVars2

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

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

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. The proof rules related to *seqDef* will be discussed in Chapter 5.

2.2.2 Undefinedness

subsec02:Undefinedness

KeY employs two ways of handling undefinedness:

1. A default value within the range of the function is chosen.
   For example $bsum\{vi\}(1,0,s)$ evaluates to 0 (regardless of s).
2. An unknown value from the range of the function is chosen.
   The prime example for this method, called *underspecification*, is division by 0 such that, e.g., $\frac{1}{0}$ is an arbitrary integer. For $cast_A(t)$ in case that t is not of type A, default values have been fixed, except for $A = Int$, see Figure 2.10.

Another frequently used way to deal with undefinedness is to chose 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 $cast_{Int}(c) \doteq 5 \rightarrow c \doteq 5$ is not universally valid. In case c is not of type Int the underspecified value for $cast_{Int}(c)$ could be 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 the thesis Kirsten [2013].

2.3 First-Order Logic for Java

sec02:JavaFOL

As already indicated in the introduction of this chapter, JFOL will be an instantiation of the extended classical first-order logic from Subsection 2.2 taylored 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 tpye 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 non-trival program properties. The data types of the integers and the theory of arrays are considered so elementary that they are already included here. Also the rather ad hoc data type *LocSet* of sets of memory locations will be covered here, since it is heavily used in the next chapter, Chapter 3. The data type of *Seq* of finite sequences however will extensively be treated later in Section 5.1.

2.3.1 Type Hierarchy and Signature

subsec02:JFOLSignature

The mandatory type hierarchy, $\mathscr{T}_J$, for JFOL is shown in Figure 2.3. Between $Object$ and $Null$ the class 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*.

The mandatory vocabulary Σ_J of JFOL is shown in Figure 2.4 using the same notation as in Definition 2.2.

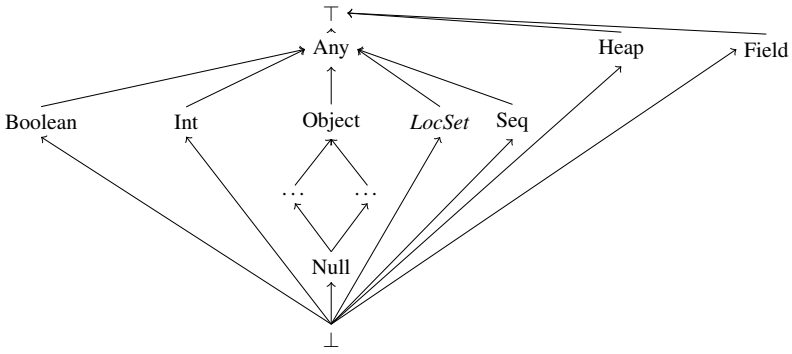

Fig. 2.3 The minimal type hierarchy $\mathscr{T}_J$ fig:typeHierarchy

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

Fig. 2.4 The vocabulary Σ_J of JFOL fig:SigmaJ

A few motivational remarks on the data type $Heap$ are in order here. 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. To model heaps it is thus necessary to have a type $Field$ in JFOL. This type is required to contain the implicit field constant *created* and the fields $arr(i)$ for array access for natural numbers $1 \leq i$. 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 to represent more functions than can possibly occur as heaps in states reacheable 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.

item:typmismatch

On one hand our heap model allows for heaps that we will never need, on the other hand this generality makes the model simpler. Simplification 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 data type $LocSet$ is a very special case of the data type of sets in that only sets of pairs (o, f) with o an object and f a field are considered. This immediately guarantees that the is-element-of relation ε is a well-founded. Problematic formulas such as $a \varepsilon a$ are already syntactically impossible.

2.3.2 Axioms

subsec02:JFOLAxioms

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	$i<j \vee i=j \vee j<i$
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	$i<j \wedge 0<k \rightarrow i*k<j*k$
less_base	$\neg(i<i)$		

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

Fig. 2.5 Integer rules

fig:intrules

Fig. 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. Only multiply_inEq0 $i \leq j \wedge 0 \leq k \rightarrow i*k \leq j*k$ does. But, multiply_inEq can be derived, although by a rather lengthy proof (65 steps) based on a normal form transformation.

Incompleteness

Mathematically the integers $(\mathbb{Z}, +, *, 0, 1, <)$ are a commutative ordered ring satisfying the wellfoundedness property: every non-empty subset of the positive integers has a least element. Wellfoundedness is a second-order property. It is approximated by the first-order induction schema, which can be interpreted to say that every non-empty 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 are still so arcane that we need not worry about this imperfection.

The figure also lists in front of each axiom the name of the taclet that implements it. These taclets allow to replace the lefthand side of an equation by its righthand side anywhere in the sequent. Taclets of this form are called *rewrite taclets* . See also Figure 2.7.

PHS: give reference to Chapt. 4 when it becomes available

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.

The meaning of the relation symbols $instance_H(x)$, $exactInstance_H(x)$ and the function symbols $cast_H(x)$ is given by the axioms in Figure 2.6. This time we present the axioms in mathematical notation for conciseness. The first axiom scheme, (Ax I), shows that adding $instance_A$ does not increase the expressive power. These predicates can be defined already in the basic logic. The same is true for the $cast_A$ functions, these functions can be defined in basic logic plus underspecification.

$$\forall x; (instance_A(x) \leftrightarrow \exists y; (y \doteq x)) \text{ with } y : A \qquad \text{(Ax-I)} \qquad \texttt{eq:ax-i}$$

$$\forall x; (exactInstance_A(x) \rightarrow instance_A(x)) \qquad \text{(Ax-E}_1\text{)} \qquad \texttt{eq:ax-e1}$$

$$\forall x; (exactInstance_A(x) \rightarrow \neg instance_B(x)) \text{ with } A \not\sqsubseteq B \qquad \text{(Ax-E}_2\text{)} \qquad \texttt{eq:ax-e2}$$

$$\forall x; (\quad instance_A(x) \rightarrow cast_A(x) \doteq x \wedge \quad \neg instance_A(x) \rightarrow cast_A(x) \doteq default_A) \qquad \text{(Ax-C)} \qquad \texttt{eq:ax-c}$$

Fig. 2.6 Axioms for type related functions `fig:AxiomsTypRelated`

The axiomatisation of the data type *Heap*, shown in Fig. 2.7, follows the pattern well known from the theory of arrays. The standard reference is [McCarthy, 1962]. There are some changes however. The implicit field *created* gets special treatment. The value of this field cannot be manipulated by the *store* function. Rule **selectOfStore** entails, assuming extensionality of heaps, that $store(h, o, created, x) = h$. The

created field of a heap can only be changed by the *create* function. This ensures that the value of the *created* field can never be changed from $TRUE$ to $FALSE$.

$$\mathsf{selectOfStore}\ \frac{\text{if } o \doteq o2 \wedge f \doteq f2 \wedge f \neq created \text{ then } cast_A(x) \text{ else } select_A(h,o2,f2)}{select_A(store(h,o,f,x),o2,f2)}\,\mathrm{R}$$

$$\mathsf{selectOfCreate}\ \frac{\text{if } o \doteq o2 \wedge o \neq null \wedge f \doteq created \text{ then } cast_A(TRUE) \text{ else } select_A(h,o2,f)}{select_A(create(h,o),o2,f)}\,\mathrm{R}$$

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

Fig. 2.7 Rules for the theory of arrays

fig:ArrayTheoryrules

A patiently explained example for the use of *store* and *select* functions can be found in Subsection 15.2.3 on page 488. While SMT solvers can handle expressions containing many occurences 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 16.1 on page 502.

The rules in Figure 2.7 are called *rewrite rules* We use an R at the right end of the fraction line to distinguish them from the sequent rules in Figures 2.1 and 2.2. A rewrite rule $\frac{t}{s}\,\mathrm{R}$ is shorthand for a sequent rule $\frac{\Gamma' \Longrightarrow \Delta'}{\Gamma \Longrightarrow \Delta}$ where $\Gamma' \Longrightarrow \Delta'$ arises from $\Gamma \Longrightarrow \Delta$ be replacing one or more occurences of the term s by t.

The rules for the data type *LocSet* are displayed in Figure 2.8. The only constraint on the membership relation ε is formulated in rule $\mathsf{equalityToElementOf}$. 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 contraint 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 $infiniteUnion\{av\}(t1)$.

2.3.3 Semantics

subsec02:JFOLSemantics

As already remarked at the start of Subsection 2.1.3, 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 in Subsection 2.3.1.

The definition of a structure $\mathscr{M}$ for a given signature in Subsection 2.1.3 was deliberatly 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. All JFOL structures $\mathscr{M} = (D, \delta, I)$ share the same *universe* or *domain* (D, δ) for a given type hierarchy $\mathscr{T}$. If the type hierarchy is extendend, the universe may also be extended. A decisive difference to the semantics from Section 2.1.3 is that now the interpretation of some symbols, types, functions, predicates, may be

$$\mathsf{equalityToElementOf}\ \frac{\forall o;\forall f;(\varepsilon(o.f.s1)\leftrightarrow\varepsilon(o.f.s2))}{s1\doteq s2}\mathrm{R}$$

$$\mathsf{elementOfEmpty}\ \frac{\Gamma\Longrightarrow\Delta}{\Gamma,\neg\varepsilon(o1,f1,empty)\Longrightarrow\Delta}$$

$$\mathsf{elementOfAllLocs}\ \frac{\Gamma\Longrightarrow\Delta}{\Gamma,\varepsilon(o1,f1,allLocs)\Longrightarrow\Delta}$$

$$\mathsf{elementOfSingleton}\ \frac{o1\doteq o2\ \&\ f1\doteq f2}{\varepsilon(o1,f1,singleton(o2,f2)}\mathrm{R}$$

$$\mathsf{elementOfUnion}\ \frac{\varepsilon(o1,f1,t1)\mid\varepsilon(o1,f1,t2)}{\varepsilon(o1,f1,union(t1,t2))}\mathrm{R}$$

$$\mathsf{elementOfIntersect}\ \frac{\varepsilon(o1,f1,t1)\ \&\ \varepsilon(o1,f1,t2)}{\varepsilon(o1,f1,intersect(t1,t2))}\mathrm{R}$$

$$\mathsf{elementOfAllFields}\ \frac{o1\doteq o2}{\varepsilon(o1,f1,allField(o2)}\mathrm{R}$$

$$\mathsf{elementOfSetMinus}\ \frac{\varepsilon(o1,f1,t1)\ \&\ \neg\varepsilon(o1,f1,t2)}{\varepsilon(o1,f1,setMinus(t1,t2))}\mathrm{R}$$

$$\mathsf{elementOfAllObjects}\ \frac{f1\doteq f2}{\varepsilon(o1,f1,allObjects(f2)}\mathrm{R}$$

$$\mathsf{elementOfInfiniteUnion}\ \frac{\exists av;\varepsilon(o1,f1,s1)}{\varepsilon(o1,f1,infiniteUnion\{av\}(t1))}\mathrm{R}$$

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

Fig. 2.8 Rules for data type *LocSet* fig:LocSetrules

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 model. Other symbols are only loosely constraint, e.g. *length* is only required to be non-negative. Completely fixed symbols will be called *rigid* as opposed to *flexible* or *non-rigid* symbols. The semantic constraints on the symbols from Figure 2.3 are shown in Figure 2.9. The

- $D^{Int}=\mathbb{Z}$,
- $D^{Boolean}=\{true,false\}$,
- $D^{T}=$ a fixed infinite set of potential instances for every T with $Null\sqsubset T\sqsubseteq Object$,
- $D^{Null}=\{\text{null}\}$,
- $D^{Heap}=$ the set of all functions $h:D^{Object}\times D^{Field}\to D^{Any}$,
- $D^{LocSet}=$ the set of all subsets of $\{(o,f)\mid o\in D^{Object}\text{ and }f\in D^{Field}\}$,
- D^{Field} a fixed infinite set that contains at least an object f^{D} for every field f occuring in the Java program under investigation.

Fig. 2.9 Semantics on type domains fig:FixedTypedomains

semantics of *Seq* will be given in Chapter 5.

Constant Domain

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

All symbols listed in Figure 2.4 are rigid. All JFOL structures $\mathscr{M} = (M, \delta, I)$ are required to satisfy the fixed interpretation put forth in Figure 2.10.

The integer operations are defined as usual with the following version of integer division:

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

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

$$mod(N,D) = N - (N/D) * D$$

Note, that this implies $mod(N,0) = N$ due to underspecification.

The semantics for the vocabulary from Figure 2.4 is shown in Figure 2.10. A few items are worth mentioning. 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 field f. This liberality necessitates the use of the $cast_A$ functions in the semantics of $select_A$. The value of the $default_A$ constant is fixed for most types A, but $default_{Int}$ is not. In this case the uninterpreted constant $default_{Int}$ implements the underspecification paradigm.

The function $anon^{\mathscr{M}}(h_1, s, h_2)^{\mathscr{M}}$ overwrites the function h_1 for the arguments $(o, f) \in s$ by the values of h_2. Since the field *created* is only allowed to be changed

1. $TRUE^{\mathcal{M}} = true$ and $FALSE^{\mathcal{M}} = false$.
2. $select_A^{\mathcal{M}}(h,o,f) = cast_A^{\mathcal{M}}(h(o,f))$
item:Sem-store
3. For arguments h,o,f,x of appropriate types the function value $h^* = store^{\mathcal{M}}(h,o,f,x)$, which is itself a function, is given by
$$h^*(o',f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq created^{\mathcal{M}} \\ h(o',f') & \text{otherwise} \end{cases}$$
item:Sem-create
4. For arguments h,o of appropriate types the function value $h^* = create^{\mathcal{M}}(h,o)$ is given by
$$h^*(o',f) = \begin{cases} true & \text{if } o' = o, o \neq \text{null and } f = created^{\mathcal{M}} \\ h(o',f) & \text{otherwise} \end{cases}$$
item:Sem-arr
5. $arr^{\mathcal{M}}$ is an injective function from $\mathbb{Z}$ into $Field^{\mathcal{M}}$,
$created^{\mathcal{M}}$ and $f^{\mathcal{M}}$ for each Java field are pairwise different elements in $Field^{\mathcal{M}}$ and also not in the range of $arr^{\mathcal{M}}$.
6. $null^{\mathcal{M}} = \text{null}$.
item:Sem-cast
7. $cast_A^{\mathcal{M}}(o) = \begin{cases} o & \text{if } o \in A^{\mathcal{M}} \\ default_A & \text{otherwise} \end{cases}$
The default element $default_A$ for type A is as follows:
$$default_A = \begin{cases} \text{null} & \text{if } A \sqsubseteq Object \\ empty & \text{if } A = LocSet \\ seqEmpty & \text{if } A = Seq \\ false & \text{if } A = Boolean \end{cases}$$
item:Sem-Inst
8. $instance_A^{\mathcal{M}} = A^{\mathcal{M}} = \{o \in M \mid \delta(o) \sqsubseteq A\}$
item:Sem-exInst
9. $IexactInstance_A^{\mathcal{M}} = \{o \in M \mid \delta(o) = A\}$
10. $length^{\mathcal{M}}(o) \in \mathbb{N}$
11. $\in^{\mathcal{M}}(o,f,s)$ iff $(o,f) \in s$
12. $empty^{\mathcal{M}} = \emptyset$, $allLocs^{\mathcal{M}} = Object^{\mathcal{M}} \times Field^{\mathcal{M}}$
13. $singleton^{\mathcal{M}}(o,f) = \{(o,f)\}$
14. $subset^{\mathcal{M}}(s_1,s_2)$ iff $s_1 \subseteq s_2$
15. $disjoint^{\mathcal{M}}(s_1,s_2)$ iff $s_1 \cap s_2 = \emptyset$
16. $union^{\mathcal{M}}(s_1,s_2) = s_1 \cup s_2$, $intersect^{\mathcal{M}}(s_1,s_2) = s_1 \cap s_2$, $setMinus(s_1,s_2) = s_1 \setminus s_2$
17. $allFields^{\mathcal{M}}(o) = \{(o,f) \mid f \in Fields^{\mathcal{M}}\}$, $allObjects^{\mathcal{M}}(f) = \{(o,f) \mid o \in Objects^{\mathcal{M}}\}$
18. $arrayRange^{\mathcal{M}}(o,i,j) = \{(o,arr^{\mathcal{M}}(x) \mid x \in \mathbb{Z}, i \leq x \leq j\}$
19. $unusedLocs^{\mathcal{M}}(h) = \{(o,f) \mid o \in Objects^{\mathcal{M}}, f \in Fields^{\mathcal{M}}, o \neq null, h(o,created^{\mathcal{M}}) = false\}$
20. $anon^{\mathcal{M}}(h_1,s,h_2)^{\mathcal{M}}(o,f) = \begin{cases} h_2(o,f) & \text{if } ((o,f) \in s \text{ and } f \neq created^{\mathcal{M}}) \\ & \quad \text{or } (o,f) \in unusedLocs^{\mathcal{M}}(h_1) \\ h_1(o,f) & \text{else} \end{cases}$

Fig. 2.10 Semantics for the vocabulary from Figure 2.4 fig:JFOLstructures

by the *create* function, *anon* should never overwrite arguments $(o, created)$. But it may overwrite h_1 for arguments (o,f) with $o \neq null$ and o not yet created should not be touched. The *anon* operation is typically applied with h_2 an unknown function, and we say that h_1 is anonymized for the locations in s.

lem:TypeRelSoundComplete **Lemma 2.24.** *The axioms in Figure 2.6 are sound and complete with respect to the given semantics.*