Wolfgang Ahrendt, Bernhard Beckert,
Richard Bubel, Reiner Hähnle,
Peter H. Schmitt, Mattias Ulbrich
(Editors)

# Deductive Software Verification— The KeY Book

## From Theory to Practice

Springer

# 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 $A \in \mathrm{TSym}$.

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 $f : A_1 \times \ldots \times A_n \to A$ 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 $A$,
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 $\perp$. 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 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 $\Sigma$ a signature for $\mathscr{T}$. The set $\mathrm{Trm}_A$ of *terms of type* $A$, for $A \neq \perp$, 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 $f : A_1 \times \ldots \times A_n \to A \in \mathrm{FSym}$ 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 $c : \to A \in \mathrm{FSym}$. 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 $\alpha$.

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 $\Sigma$ 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 $1 \le i \le n$.
   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 $\phi_1$ then $\phi_2$ else $\phi_3$) 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 $\Sigma$ and $\mathscr{T}$ explicit by writing $\mathrm{Trm}_{A,\Sigma}$, $\mathrm{Fml}_{\Sigma}$ 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 $\phi$ we define the sets $var(t)$, $var(\phi)$ of all variables occurring in $t$ or $\phi$ and the sets $fv(t)$, $fv(\phi)$ of all variables with at least one free occurrence in $t$ or $\phi$:

| | | | | |
|---|---|---|---|---|
| $var(v) =$ | $\{v\}$ | $fv(v) =$ | $\{v\}$ | 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 $\phi$ 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 $\phi = p(t_1,\ldots,t_n)$ |
| $var(\neg\phi) =$ | $var(\phi)$ | $fv(\neg\phi) =$ | $fv(\phi)$ | |
| $var(\phi) =$ | $var(\phi_1) \cup var(\phi_2)$ | $fv(\phi) =$ | $fv(\phi_1) \cup fv(\phi_2)$ | 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*, 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* $\tau$ 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 $\tau$ is called a *ground substitution* if $\tau(v)$ is a ground term for all $v \in \mathrm{dom}(\tau)$.

We will only encounter substitutions $\tau$ 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 $\tau$. It remains to make precise how a substitution $\tau$ is applied to terms and formulas.

**Definition 2.7.** Let $\tau$ 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 $\tau$ if $x \in \mathrm{dom}(\tau)$
3. $\tau(f(t_1,\dots,t_k)) = f(\tau(t_1),\dots,\tau(t_k))$ if $t = f(t_1,\dots,t_k)$

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

4. $\tau(\mathit{true}) = \mathit{true}$, $\tau(\mathit{false}) = \mathit{false}$
5. $\tau(p(t_1,\dots,t_k)) = p(\tau(t_1),\dots,\tau(t_k))$ if $\phi$ is the atomic formula $p(t_1,\dots,t_k)$
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 $\phi$ of the quantified formula is left untouched. This is effected by removing $v$ from the domain of $\tau$.

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 $\mathit{int} \in \mathrm{TSym}$, function symbols $+ : \mathit{int} \times \mathit{int} \to \mathit{int}$, $* : \mathit{int} \times \mathit{int} \to \mathit{int}$, $- : \mathit{int} \to \mathit{int}$, $\mathit{exp} : \mathit{int} \times \mathit{int} \to \mathit{int}$ 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 furthermore $x : \mathit{int}$, $y : \mathit{int}$ be variables of sort $\mathit{int}$. 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 nonground substitution $\tau_2 = [y/x]$ on $\phi_3$ 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 $\tau$ is said to create a *clash* with formula $\phi$ if a variable $w$ in a term $\tau(v)$ for $v \in \text{dom}(\tau)$ ends up in the scope of a quantifier $Qw$ in $\phi$. For $\tau_2$ 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 $\phi$ be a formula that contains somewhere an occurrence of the term $t_1$. How should we refer to the formula arising from $\phi$ by replacing $t_1$ by $t_2$? E.g. replace $2xy$ in $\phi_2$ by $xy2$. The solution is to use a new variable $z$ and a formula $\phi_0$ 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 $\phi_i$ and $\psi_j$ 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 $\Gamma, \Delta$ for set of formulas, $\psi, \phi$ for formulas and $t, c$ for terms and constants. We use $\Gamma, \phi$ and $\psi, \Delta$ to stand for $\Gamma \cup \{\phi\}$ and $\{\psi\} \cup \Delta$. An instance of a rule is obtained by consistently replacing the

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

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

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

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

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

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

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

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

**Fig. 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 $\Longrightarrow p \wedge q \rightarrow q \wedge p$. 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 $p \wedge q \rightarrow q \wedge p$ is a tautology. Application of the rule impRight reduces our proof goal to $p \wedge q \Longrightarrow q \wedge p$ and application of andLeft further to $p, q \Longrightarrow q \wedge p$. 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

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{p,q \Longrightarrow q} \qquad \dfrac{*}{p,q \Longrightarrow p}}{p,q \Longrightarrow q \wedge p}}{p \wedge q \Longrightarrow q \wedge p}}{\Longrightarrow p \wedge q \rightarrow q \wedge p}$$

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 $\exists v; \forall w; p(v,w) \rightarrow \forall w; \exists v; p(v,w)$:

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

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. 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 $a : \rightarrow A$ and $b : \rightarrow B$, 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.

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

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

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

**Fig. 2.2** Equality rules for the logic FOL

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

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 $\phi$ 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 $\phi$ 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 $\phi$ 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 $\Sigma$ consists of

1. a set $D$,

2. a typing function $\delta : D \to \text{TSym} \setminus \{\perp\}$ such that for every $A \in \text{TSym}$ 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 $A, B \in \text{TSym} \setminus \{\perp\}$ there is an element $o \in D^A \cap D^B$ only if there exists $C \in \text{TSym}$, $C \neq \perp$ with $C \sqsubseteq A$ and $C \sqsubseteq B$.

**Lemma 2.12.** *The type domains for a universe $(D, \delta)$ 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 $\Sigma$ consists of

- a domain $(D, \delta)$,
- 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 $c : \ \to A \in \text{FSym}$ 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 $\mathscr{T}$-$\Sigma$-structure.

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

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

For a variable assignment $\beta$, a variable $v : A \in \text{VSym}$ 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 $\Sigma$, and variable assignment $\beta$ by mutual recursion.

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

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

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

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

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

For a 0-place predicate symbol $p$, clause (2) says $\mathscr{M} \models p$ 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 $(\mathscr{M},\beta) \models t_1 \doteq t_2$ iff $\mathrm{val}_{\mathscr{M},\beta}(t_1) = \mathrm{val}_{\mathscr{M},\beta}(t_2)$.

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

If $\phi$ is a formula without free variables, we may write $\mathscr{M} \models \phi$ since the variable assignment $\beta$ 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 $\mathscr{T}_2 = (\mathrm{TSym}_2, \sqsubseteq_2)$ 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 $\Delta$ 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, $\perp \sqsubseteq_2 A \sqsubseteq_2 \top$ 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* $\mathscr{T}_2 = (\mathrm{TSym}_2, \sqsubseteq_2)$ *be an extension of* $\mathscr{T}_1 = (\mathrm{TSym}_1, \sqsubseteq_1)$ *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. □

**Definition 2.19.** Let $\mathscr{T}$ be a type hierarchy and $\Sigma$ 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. $\phi$ is a *logical consequence* of $\Phi$, in symbols $\Phi \models \phi$, if for all type hierarchies $\mathscr{T}'$ with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and all $\mathscr{T}'$-$\Sigma$-structures $\mathscr{M}$ such that $\mathscr{M} \models \Phi$, also $\mathscr{M} \models \phi$ holds.
2. $\phi$ is *universally valid* if it is a logical consequence of the empty set, i.e., if $\emptyset \models \phi$.
3. $\phi$ is satisfiable if there is a type hierarchy $\mathscr{T}'$, with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and a $\mathscr{T}'$-$\Sigma$-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 $\Phi$, it can easily be seen that:

- $\Phi \models \phi$ iff the formula $\bigwedge \Phi \to \phi$ is universally valid.
- $\phi$ 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 $\mathscr{T}$-$\Sigma$-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 $\phi$ depends on symbols that do not occur in $\phi$ 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* $\Sigma$ *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* $\phi$ *is universally valid*
  *i.e., FOL is sound.*
- *if* $\phi$ *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 $\phi_b$ containing binder symbols there is a formula $\phi$ without such that $\phi_b$ is universally valid if and only if $\phi$ 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 $\mathrm{Trm}_{int}$, then $bsum\{vi\}(b_0, b_1, s)$ is in $\mathrm{Trm}_{int}$.
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 $\mathrm{Trm}_{int}$, then $bprod\{vi\}(b_0, b_1, s)$ is in $\mathrm{Trm}_{int}$.
6. If $vi$ is a variable of arbitrary type and $s$ a term of type *LocSet*, then $infiniteUnion\{vi\}(s)$ is in $\mathrm{Trm}_{LocSet}$.
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 $\mathrm{Trm}_{any}$, then $seqDef\{vi\}(b_0, b_1, s)$ is in $\mathrm{Trm}_{Seq}$.

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

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

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 $\Sigma_{b_0 \le vi < b_1} s_{vi}$ 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. 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

$seqDef\{vi\}(b_0, b_1, s)$ 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., $seqDef\{vi\}(b_0, b_1, s) = \langle\rangle$. 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 $val_{\mathscr{M}}(f)$ at position $\bar{a}$ is set to a default within the intended range of $f$. E.g., $bsum\{vi\}(1, 0, s)$ evaluates to 0 (regardless of $s$).
2. The value of the function $val_{\mathscr{M}}(f)$ 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 $val_{\mathscr{M}_b}(f)(\bar{a}) = b$. 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 $cast_{int}(c) \doteq 5 \rightarrow c \doteq 5$ is not universally valid. In case $c$ is not of type *int* the underspecified value for $cast_{int}(c)$ could be 5 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 $\Sigma$ 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 $\Sigma_J$ 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 $\mathscr{T}_J$ 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*.

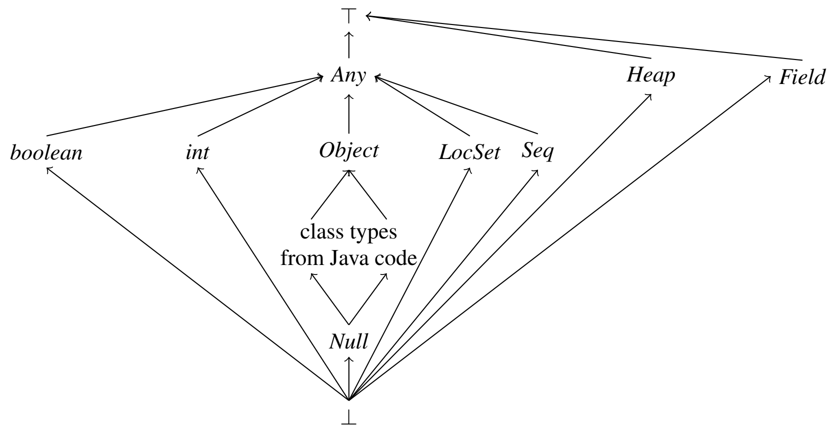


**Fig. 2.3** The mandatory type hierarchy $\mathscr{T}_J$ of JFOL

The mandatory vocabulary $\Sigma_J$ 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 $\Sigma$ may be a strict superset of $\Sigma_J$. 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 | $null : Null$ |
| | $length : Object \to int$ |
| | $cast_A : Object \to A$ for any $A$ in $\mathscr{T}$ with $\perp \sqsubset A \sqsubseteq Object$. |
| | $instance_A : Any \to boolean$ for any type $A \sqsubseteq Any$ |
| | $exactInstance_A : Any \to boolean$ for any type $A \sqsubseteq Any$ |
| *Field* | $created : Field$ |
| | $arr : int \to Field$ |
| | $f : Field$ for every Java field $f$ |
| *Heap* | $select_A : Heap \times Object \times Field \to A$ for any type $A \sqsubseteq Any$ |
| | $store : Heap \times Object \times Field \times Any \to Heap$ |
| | $create : Heap \times Object \to Heap$ |
| | $anon : Heap \times LocSet \times Heap \to Heap$ |
| | $wellFormed(Heap)$ |
| *LocSet* | $\varepsilon(Object, Field, LocSet)$ |
| | $empty, allLocs : LocSet$ |
| | $singleton : Object \times Field \to LocSet$ |
| | $subset(LocSet, LocSet)$ |
| | $disjoint(LocSet, LocSet)$ |
| | $union, intersect, setMinus : LocSet \times LocSet \to LocSet$ |
| | $allFields : Object \to LocSet,\ allObjects : Field \to LocSet$ |
| | $arrayRange : Object \times int \times int \to LocSet$ |
| | $unusedLocs : Heap \to LocSet$ |

**Fig. 2.4** The mandatory vocabulary $\Sigma_J$ 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 | $i < j \wedge j < k \to i < k$ | 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 | $i < j \to i + k < j + k$ | multiply_inEq | $i < j \wedge 0 < k \to i * k < j * k$ |

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

**Fig. 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 $i \leq j \wedge 0 \leq k \to i * k \leq j * k$ 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
$select_A(store(h,o,f,x),o2,f2) \rightsquigarrow \text{if } o \doteq o2 \wedge f \doteq f2 \text{ then } x \text{ else } select_A(h,o2,f2)$.
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 $select_A(store(h,o,f,x),o2,f2) \rightsquigarrow$
$\text{if } o \doteq o2 \wedge f \doteq f2 \wedge f \dot{\neq} created \text{ then } cast_A(x) \text{ else } select_A(h,o2,f2)$

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

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

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

**Fig. 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
$wellFormed(h) \wedge o \not\doteq null \wedge select_B(h,o,arr(i)) \rightarrow select_A(h,o,arr(i))$
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*

**Fig. 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

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

**Fig. 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 $\varepsilon$ 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 $\varepsilon$ 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 $\varepsilon$. 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 | $\varepsilon(o1, f1, \mathit{singleton}(o2, f2)$ | $\rightsquigarrow o1 \doteq o2 \wedge f1 \doteq f2$ |
| elementOfUnion | $\varepsilon(o1, f1, \mathit{union}(t1, t2))$ | $\rightsquigarrow \varepsilon(o1, f1, t1) \vee \varepsilon(o1, f1, t2)$ |
| subsetToElementOf | $\mathit{subset}(t1, t2)$ | $\rightsquigarrow \forall o; \forall f; (\varepsilon(o, f, t1) \rightarrow \varepsilon(o, f, t2))$ |
| elementOfIntersect | $\varepsilon(o1, f1, \mathit{intersect}(t1, t2))$ | $\rightsquigarrow \varepsilon(o1, f1, t1) \wedge \varepsilon(o1, f1, t2)$ |
| elementOfAllFields | $\varepsilon(o1, f1, \mathit{allFields}(o2)$ | $\rightsquigarrow o1 \doteq o2$ |
| elementOfSetMinus | $\varepsilon(o1, f1, \mathit{setMinus}(t1, t2))$ | $\rightsquigarrow \varepsilon(o1, f1, t1) \wedge \neg\varepsilon(o1, f1, t2)$ |
| 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.

**Fig. 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 $length^{\mathscr{M}}(o) = n$. 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 $length^{\mathscr{M}}(o) = n$.
- $D^{Null} = \{null\}$,
- $D^{Heap}$ = 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.

**Fig. 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 $\mathscr{M} \models \phi$ 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 $\mathscr{M} \models \neg\phi$.

The interpretation of all the JFOL function and predicate symbols listed in Figure 2.4 is at least partly fixed. All JFOL structures $\mathscr{M} = (M, \delta, I)$ 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. $TRUE^{\mathscr{M}} = tt$ and $FALSE^{\mathscr{M}} = ff$
2. $select_A^{\mathscr{M}}(h,o,f) = cast_A^{\mathscr{M}}(h(o,f))$
3. $store^{\mathscr{M}}(h,o,f,x) = h^*$, where the function $h^*$ is defined by

$$h^*(o',f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq created^{\mathscr{M}} \\ h(o',f') & \text{otherwise} \end{cases}$$

4. $create^{\mathscr{M}}(h,o) = h^*$, where the function $h^*$ is defined by

$$h^*(o',f) = \begin{cases} tt & \text{if } o' = o, o \neq null \text{ and } f = created^{\mathscr{M}} \\ h(o',f) & \text{otherwise} \end{cases}$$

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

$$h^*(o,f) = \begin{cases} h_2(o,f) & \text{if } (o,f) \in s \text{ and } f \neq created^{\mathscr{M}}, \text{ or} \\ & \quad (o,f) \in unusedLocs^{\mathscr{M}}(h_1) \\ h_1(o,f) & \text{otherwise} \end{cases}$$

**Fig. 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:

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

**Fig. 2.12** Semantics for the predicate *wellFormed*

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

Thus integer division is a total function with arbitrary values for $x/^{\mathscr{M}}0$. Division is an example of a partially fixed 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

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

Note, that this implies $\mathrm{mod}(n,0) = n$ 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].

# References


Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996. (Cited on page 240.)

Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, 2008, San Jose, CA, USA*, pages 335–348. USENIX Association, 2008. (Cited on page 606.)

Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Proceedings of the 8th European Workshop on Logics in Artificial Intelligence (JELIA)*, volume 1919 of *LNCS*, pages 21–36. Springer, October 2000. (Cited on page 13.)

Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In Geoff Sutcliff and Andrei Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*, pages 412–426. Springer, December 2005. (Cited on pages 12 and 64.)

Wolfgang Ahrendt, Richard Bubel, and Reiner Hähnle. Integrated and tool-supported teaching of testing, debugging, and verification. In Jeremy Gibbons and José Nuno Oliveira, editors, *Second International Conference on Teaching Formal Methods, Proceedings*, volume 5846 of *LNCS*, pages 125–143. Springer, 2009a. (Cited on page 7.)

Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands. Proceedings*, volume 5850 of *LNCS*, pages 612–627, 2009b. (Cited on page 56.)

Wolfgang Ahrendt, Wojciech Mostowski, and Gabriele Paganelli. Real-time Java API specifications for high coverage test generation. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 145–154, New York, NY, USA, 2012. ACM. (Cited on pages 6 and 609.)

Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A specification language for static and runtime verification of data and control properties. In Nikolaj Bjørner and Frank de Boer, editors, *Formal Methods - 20th International Symposium, Oslo, Norway, Proceedings*, volume 9109 of *LNCS*, pages 108–125. Springer, 2015. (Cited on page 519.)

Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test data generation of bytecode by CLP partial evaluation. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR, Valencia, Spain, Revised Selected Papers*, volume 5438 of *LNCS*, pages 4–23. Springer, 2009. (Cited on page 4.)

Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, and Guillermo Román-Díez. Verified resource guarantees for heap manipulating programs. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia. Proceedings*, volume 7212 of *LNCS*. Springer, 2012. (Cited on page 4.)

Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE, Edinburgh, UK*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010. (Cited on page 9.)

Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA*, pages 71–82. ACM, 2012. (Cited on pages 3, 240, 241 and 377.)

Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of concurrent systems with VerCors. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, Advanced Lectures*, volume 8483 of *LNCS*, pages 172–216. Springer, 2014a. (Cited on page 350.)

Afshin Amighi, Stefan Blom, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Formal specifications for Java's synchronisation classes. In Alberto Lluch Lafuente and Emilio Tuosto, editors, *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy*, pages 725–733. IEEE Computer Society, 2014b. (Cited on pages 3 and 378.)

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. (Cited on pages 421 and 448.)

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 91–102. ACM, 2006. (Cited on page 454.)

Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. (Cited on page 448.)

Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting dependences and interactions in feature-oriented design. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA*, pages 161–170. IEEE Computer Society, 2010. (Cited on page 17.)

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, 2014*, pages 1083–1094. ACM, 2014. (Cited on page 450.)

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. (Cited on page 412.)

Thomas Baar. Metamodels without metacircularities. *L'Objet*, 9(4):95–114, 2003. (Cited on page 2.)

Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of dynamic logic for modelling OCL's @pre operator. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, Revised Papers*, volume 2244 of *LNCS*, pages 47–54. Springer, 2001. (Cited on page 249.)

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In Thomas S. E. Maibaum, editor, *Fundamental Approaches to*

*Software Engineering, Third Internationsl Conference, FASE 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany. Proceedings*, volume 1783 of *LNCS*, pages 363–366. Springer, 2000. (Cited on pages 10, 239 and 353.)

Anindya Banerjee, Michael Barnett, and David A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada. Proceedings*, volume 5295 of *LNCS*, pages 177–191, New York, NY, 2008a. Springer. (Cited on page 350.)

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, Proceedings*, volume 5142 of *LNCS*, pages 387–411, New York, NY, 2008b. Springer. (Cited on page 350.)

Michael Bär. Analyse und Vergleich verifizierbarer Wahlverfahren. Diplomarbeit, Fakultät für Informatik, KIT, 2008. (Cited on page 606.)

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustin M. Leino, and Wolfgang Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6): 27–56, 2004. (Cited on pages 210, 215 and 348.)

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006. (Cited on pages 10 and 216.)

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), International Workshop, Marseille, France, Revised Selected Papers*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005a. (Cited on pages 241 and 348.)

Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. In *ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs*, pages 51–60, January 2005b. (Cited on page 210.)

Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Communications ACM*, 54(6): 81–91, 2011. (Cited on page 349.)

Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. (Cited on pages 12 and 18.)

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17), Pacific Grove, CA, USA*, pages 100–114, Washington, USA, 2004. IEEE CS. (Cited on page 454.)

Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany*, pages 86–95. IEEE Computer Society, 2005. (Cited on page 230.)

Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK: A tool for validation of security and behaviour of Java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, Revised Lectures*, volume 4709 of *LNCS*, pages 152–174, Berlin, 2007. Springer. (Cited on page 240.)

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the*

*36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA*, pages 90–101. ACM, January 2009. (Cited on page 607.)

Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland. Proceedings*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011. (Cited on page 483.)

Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, 2013*, pages 123–134. ACM, 2013a. (Cited on page 5.)

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Transactions on Programming Languages and Systems*, 35(3):9, 2013b. (Cited on page 607.)

Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010. Version 1.5. (Cited on page 241.)

Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia*, volume 102 of *EPTCS*, pages 18–32, 2012. (Cited on page 2.)

Kent Beck. *JUnit Pocket Guide: quick lookup and advice*. O'Reilly, 2004. (Cited on pages 416 and 421.)

Tobias Beck. Verifizierbar korrekte Implementierung von Bingo Voting. Studienarbeit, Fakultät für Informatik, KIT, March 2010. (Cited on page 606.)

Bernhard Beckert and Daniel Bruns. Formal semantics of model fields in annotation-based specifications. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence - 35th Annual German Conference on AI, Saarbrücken, Germany. Proceedings*, number 7526 in LNCS, pages 13–24. Springer, 2012. (Cited on page 310.)

Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland. Revised Papers*, volume 4454 of *LNCS*, pages 207–216. Springer, 2007. (Cited on page 416.)

Bernhard Beckert and Sarah Grebing. Evaluating the usability of interactive verification systems. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, 2012*, volume 873 of *CEUR Workshop Proceedings*, pages 3–17. CEUR-WS.org, 2012. (Cited on page 8.)

Bernhard Beckert and Reiner Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1): 20–29, Jan.–Feb. 2014. (Cited on pages 2, 3 and 18.)

Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006. (Cited on page 64.)

Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzé, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, 2003. (Cited on pages 354 and 376.)

Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006. (Cited on page 65.)

Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated*

*Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK. Proceedings*, volume 2999 of *LNCS*, pages 207–226. Springer, 2004. (Cited on page 51.)

Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005. (Cited on page 51.)

Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1):17–53, 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence. (Cited on page 11.)

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in LNCS. Springer, 2007. (Cited on pages ix, 16, 230, 240, 272, 306, 376, 384, 527 and 576.)

Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung. The KeY approach for the cryptographic verification of Java programs: A case study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012. (Cited on page 594.)

Bernhard Beckert, Thorsten Bormer, and Markus Wagner. A metric for testing program verification systems. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs. Seventh International Conference, TAP 2013, Budapest, Hungary*, volume 7942 of *LNCS*, pages 56–75. Springer, 2013. (Cited on page 65.)

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, Revised Selected Papers*, number 8901 in LNCS, pages 19–37. Springer, 2014. (Cited on page 460.)

Bernhard Beckert, Vladimir Klebanov, and Mattias Ulbrich. Regression verification for Java using a secure information flow calculus. In Rosemary Monahan, editor, *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic*, pages 6:1–6:6. ACM, 2015. (Cited on page 428.)

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving: Second International Conference, ITP 2011, Berg en Dal, The Netherlands. Proceedings*, pages 22–38. Springer, 2011. (Cited on page 316.)

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 14–25. ACM, 2004. (Cited on pages 5, 483 and 607.)

Dirk Beyer. Software verification and verifiable witnesses — (report on SV-COMP 2015). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK. Proceedings*, volume 9035 of *LNCS*, pages 401–416. Springer, 2015. (Cited on pages 4 and 18.)

Joshua Bloch. *Effective Java: Programming Language Guide*. The Java Series. Addison-Wesley, 2nd edition, 2008. (Cited on page 261.)

Arjan Blom, Gerhard de Koning Gans, Erik Poll, Joeri de Ruiter, and Roel Verdult. Designed to fail: A USB-connected reader for online banking. In Audun Jøsang and Bengt Carlsson, editors, *Secure IT Systems - 17th Nordic Conference, NordSec 2012, Karlskrona, Sweden. Proceedings*, volume 7617 of *LNCS*, pages 1–16. Springer, 2012. (Cited on page 353.)

Stefan Blom and Marieke Huisman. The VerCors Tool for verification of concurrent programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore. Proceedings*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014. (Cited on page 378.)

Jens-Matthias Bohli, Christian Henrich, Carmen Kempka, Jörn Müller-Quade, and Stefan Röhrich. Enhancing electronic voting machines on the example of Bingo voting. *IEEE Transactions on Information Forensics and Security*, 4(4):745–750, 2009. (Cited on page 606.)

Greg Bollella and James Gosling. The real-time specification for Java. *IEEE Computer*, pages 47–54, June 2000. (Cited on page 5.)

Alex Borgida, John Mylopoulos, and Raymond Reiter. "…And nothing else changes": On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10): 785–798, 1995. (Cited on pages 233 and 321.)

Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, editors, *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada*, pages 70–78. IEEE Computer Society, May 2009. (Cited on page 449.)

Raymond T. Boute. Calculational semantics: Deriving programming theories from equations by functional predicate calculus. *ACM Transactions on Programming Languages and Systems*, 28 (4):747–793, 2006. (Cited on page 574.)

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975. (Cited on page 383.)

John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA. Proceedings*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003. (Cited on pages 378 and 379.)

Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007. (Cited on page 537.)

Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'03), Darmstadt*, number 408 in Technical Report, ETH Zürich, pages 51–60, July 2003. (Cited on page 350.)

Cees-Bart Breunesse, Néstor Cataño, Marieke Huisman, and Bart Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005. (Cited on page 226.)

Daniel Bruns. Elektronische Wahlen: Theoretisch möglich, praktisch undemokratisch. *FIfF-Kommunikation*, 25(3):33–35, September 2008. (Cited on page 594.)

Daniel Bruns. Formal semantics for the Java Modeling Language. Diploma thesis, Universität Karlsruhe, 2009. (Cited on pages 195, 215, 243, 245 and 350.)

Daniel Bruns. Specification of red-black trees: Showcasing dynamic frames, model fields and sequences. In Wolfgang Ahrendt and Richard Bubel, editors, *10th KeY Symposium*, Nijmegen, the Netherlands, 2011. Extended Abstract. (Cited on page 296.)

Richard Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, Universität Karlsruhe, 2007. (Cited on page 306.)

Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring the correctness of lightweight tactics for Java Card dynamic logic. *Electronic Notes in Theoretical Computer Science*, 199:107–128, 2008. (Cited on pages 138 and 144.)

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madeleine, editors, *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, Revised Lectures*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009. (Cited on pages x, 171, 454, 471 and 474.)

Richard Bubel, Reiner Hähnle, and Ulrich Geilmann. A formalisation of Java strings for program specification and verification. In Gilles Barthe and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay. Proceedings*, volume 7041 of *LNCS*, pages 90–105. Springer, 2011. (Cited on page x.)

Richard Bubel, Antonio Flores Montoya, and Reiner Hähnle. Analysis of executable software models. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar B. Johnsen, and Ina Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods*

*for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014a. (Cited on page 16.)

Richard Bubel, Reiner Hähnle, and Maria Pelevina. Fully abstract operation contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece*, volume 8803 of *LNCS*, pages 120–134. Springer, October 2014b. (Cited on page 9.)

Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Proceedings*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003a. (Cited on page 239.)

Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy. Proceedings*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003b. (Cited on page 353.)

Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy*, pages 443–446. IEEE Computer Society, 2008. (Cited on page 450.)

Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress '74, Stockholm*, pages 308–312. Elsevier/North-Holland, 1974. (Cited on pages 12 and 383.)

Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego, CA, USA, Proceedings*, pages 209–224. USENIX Association, 2008a. (Cited on page 450.)

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2), 2008b. (Cited on page 450.)

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA*, pages 1066–1071. ACM, 2011. (Cited on page 449.)

Néstor Cataño, Tim Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. Translating B machines to JML specifications. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy*, pages 1271–1277, New York, NY, USA, 2012. ACM. (Cited on page 240.)

Néstor Cataño and Marieke Huisman. CHASE: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA. Proceedings*, volume 2575 of *LNCS*, pages 26–40. Springer, 2003. (Cited on page 240.)

Patrice Chalin. Improving JML: For a safer and more effective language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy. Proceedings*, volume 2805 of *LNCS*, pages 440–461. Springer, 2003. (Cited on page 231.)

Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004. Special issue: ECOOP 2003 Workshop on FTfJP. (Cited on page 232.)

Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA*, pages 23–33. IEEE Computer Society, 2007. (Cited on page 286.)

Patrice Chalin and Frédéric Rioux. Non-null references by default in the Java modeling language. *SIGSOFT Software Engineering Notes*, 31(2), September 2005. (Cited on page 246.)

Patrice Chalin and Frédéric Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland. Proceedings*, volume 5014 of *LNCS*, pages 246–261. Springer, 2008. (Cited on page 286.)

Patrice Chalin, Perry R. James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *Software, IET*, 2(6):515–531, 2008. (Cited on page 246.)

Patrice Chalin, Robby, Perry R. James, Jooyong Lee, and George Karabotsos. Towards an industrial grade IVE for Java and next generation research platform for JML. *STTT*, 12(6):429–446, 2010. (Cited on page 239.)

Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany. Proceedings*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015. (Cited on page 6.)

David Chaum, Richard T. Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily (Emily Huei-Yi) Shen, Alan T. Sherman, and Poorvi L. Vora. Scantegrity II: End-to-end verifiability by voters of optical scan elections through confirmation codes. *IEEE Transactions on Information Forensics and Security*, October 2009. (Cited on page 606.)

Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, June 2000. (Cited on pages 353 and 354.)

Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, Ames, 2003. Technical Report 03-09. (Cited on page 239.)

Yoonsik Cheon. Automated random testing to detect specification-code inconsistencies. In Dimitris A. Karras, Daming Wei, and Jaroslav Zendulka, editors, *International Conference on Software Engineering Theory and Practice, SETP-07, Orlando, Florida, USA*, pages 112–119. ISRST, 2007. (Cited on page 239.)

Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-oriented Programing*, 7(6):39–49, 1994. (Cited on page 240.)

Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada*, pages 48–64, Vancouver, Canada, October 1998. ACM. (Cited on pages 13 and 348.)

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. (Cited on page 6.)

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, California, USA*, pages 354–368. IEEE Computer Society, 2008. (Cited on pages 594 and 606.)

Ellis S. Cohen. Information transmission in computational systems. In Saul Rosen and Peter J. Denning, editors, *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA*, pages 133–139. ACM, 1977. (Cited on page 454.)

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42, Berlin, August 2009. Springer. (Cited on pages 241 and 349.)

David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005. (Cited on page 350.)

David R. Cok. Adapting JML to generic types and Java 1.6. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 27–35, 2008. (Cited on pages 195 and 237.)

David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA. Proceedings*, volume 6617 of *LNCS*, pages 472–479. Springer, Berlin, 2011. (Cited on pages 239 and 426.)

David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005. (Cited on pages 195 and 240.)

David R. Cok and Gary T. Leavens. Extensions of the theory of observational purity and a practical design for JML. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 43–50, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, 2008. School of EECS, UCF. (Cited on page 210.)

Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978. (Cited on page 65.)

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. (Cited on pages 167 and 168.)

Lajos Cseppento and Zoltán Micskei. Evaluating symbolic execution-based test tools. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Society, April 2015. (Cited on page 449.)

Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Kluwer, Dordrecht, 1999. (Cited on page 11.)

Ádám Darvas and Rustin Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal. Proceedings*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007. (Cited on page 210.)

Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006. (Cited on page 210.)

Ádám Darvas and Peter Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich, 2007. (Cited on pages 195 and 243.)

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003. (Cited on pages 5 and 278.)

Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany. Proceedings*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005. (Cited on pages x, 5, 278 and 454.)

Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient well-definedness checking. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia. Proceedings*, LNCS, pages 100–115, Berlin, Heidelberg, 2008. Springer. (Cited on page 286.)

Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof pearl: The key to correct and stable sorting. *J. Automated Reasoning*, 53(2):129–139, 2014. (Cited on page 609.)

Stijn De Gouw, Jurriaan Rot, Frank S. De Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina Pasareanu, editors, *Computer Aided Verification - 27th International Conference,*

*CAV 2015, San Francisco, CA, USA. Proceedings, Part I*, volume 9206 of *LNCS*, pages 273–289. Springer, July 2015. (Cited on page 9.)

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 (5):236–243, 1976. (Cited on page 454.)

Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. *Electronic Notes in Theoretical Computer Science*, 1:91–113, 1995. This issue contains revised papers presented at the Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, (MFPS XI), Tulane University, New Orleans, 1995. Managing editors: Michael Mislove and Maurice Nivat and Christos Papadimitriou. (Cited on page 219.)

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. (Cited on pages 571 and 577.)

Crystal Chang Din. *Verification Of Asynchronously Communicating Objects*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, March 2014. (Cited on page 6.)

Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In Gail E. Harris, editor, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 213–226, New York, NY, 2008. ACM. (Cited on page 316.)

Huy Q. Do, Richard Bubel, and Reiner Hähnle. Exploit generation for information flow leaks in object-oriented programs. In Hannes Federath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany. Proceedings*, volume 455 of *LNCS*, pages 401–415. Springer, 2015. (Cited on page 17.)

Quoc Huy Do, Eduard Kamburjan, and Nathan Wasser. Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust, 5th Intl. Conf., POST, Eindhoven, The Netherlands*, volume 9635 of *LNCS*, pages 97–115. Springer, 2016. (Cited on pages x, 5 and 189.)

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. (Cited on page 595.)

Felix Dörre and Vladimir Klebanov. Pseudo-random number generator verification: A case study. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 9593 of *LNCS*. Springer, 2015. (Cited on page 455.)

Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8. (Cited on pages 2 and 108.)

Christian Engel. A translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005. (Cited on pages 195 and 243.)

Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Bertrand Meyer and Yuri Gurevich, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland. Revised Papers*, volume 4454 of *LNCS*. Springer, 2007. (Cited on pages x, 4 and 416.)

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007. (Cited on page 240.)

Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. (Cited on pages 18 and 413.)

Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *Proceedings, 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004. (Cited on page 64.)

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, ASE '14, pages 349–360. ACM, 2014. (Cited on pages 17 and 483.)

Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999. (Cited on page 609.)

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spririt of ghost code. In Armin Biere, Swen Jacobs, and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria. Proceedings*, volume 8559 of *LNCS*, pages 1–16. Springer, 2014. (Cited on page 269.)

John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. Vienna development method. In Benjamin W. Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008. (Cited on page 240.)

Cormac Flanagan and K.Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Report 2000-003, DEC-SRC, December 2000. (Cited on page 240.)

Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. (Cited on pages 194 and 234.)

M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *Computer Journal*, 14(4): 391–395, 1971. (Cited on page 609.)

Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany. Proceedings*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. (Cited on page 6.)

Stefan J. Galler and Bernhard K. Aichernig. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer*, 16(6):727–751, 2014. (Cited on page 448.)

Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987. (Cited on pages 27 and 35.)

Flavio D. Garcia, Gerhard Koning Gans, Ruben Muijrers, Peter Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE classic. In Sushil Jajodia and Javier Lopez, editors, *Proceedings of the 13th European Symposium on Research in Computer Security*, volume 5283 of *LNCS*, pages 97–114. Springer, 2008. (Cited on page 353.)

Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. In Miki Herrmann, editor, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia. Proceedings*, LNCS, pages 332–346. Springer, October 2006. (Cited on page 68.)

Ullrich Geilmann. Formal verification using Java's String class. Studienarbeit, Chalmers University of Technology and Universität Karlsruhe, November 2009. (Cited on page 161.)

Robert Geisler, Marcus Klar, and Felix Cornelius. InterACT: An interactive theorem prover for algebraic specifications. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany. Proceedings*, volume 1101 of *LNCS*, pages 563–566. Springer, 1996. (Cited on page 108.)

Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, SE-1(1):68–75, March 1975. (Cited on page 234.)

Martin Giese. Taclets and the KeY prover. In David Aspinall and Christoph Lüth, editors, *Proc. User Interfaces for Theorem Provers Workshop, UITP, Rome, 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 67–79. Elsevier, 2004. (Cited on page 108.)

Martin Giese. A calculus for type predicates and type coercion. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany. Proceedings*, volume 3702 of *LNCS*, pages 123–137. Springer, 2005. (Cited on page 35.)

Christoph Gladisch. Verification-based test case generation for full feasible branch coverage. In Antonio Cerone and Stefan Gruner, editors, *Proceedings, Sixth IEEE International Conference*

*on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa*, pages 159–168. IEEE Computer Society, 2008. (Cited on pages x, 434 and 436.)

Christoph Gladisch and Shmuel Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In Leonardo de Moura and Juliano Iyoda, editors, *Formal Methods: Foundations and Applications - 16th Brazilian Symposium, SBMF 2013, Brasilia, Brazil. Proceedings*, volume 8195 of *LNCS*, pages 99–114. Springer, 2013. (Cited on pages 296 and 300.)

Christoph David Gladisch. *Verification-based software-fault detection*. PhD thesis, Karlsruhe Institute of Technology, 2011. (Cited on pages 416 and 436.)

Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012. (Cited on page 450.)

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. (Cited on page 65.)

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982. (Cited on page 454.)

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979. (Cited on page 10.)

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2013. (Cited on pages 52, 53, 54, 55, 60, 91, 156, 197, 237, 247, 622 and 623.)

Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs – A practical guide. In Stefan Wagner and Horst Lichter, editors, *Software Engineering (Workshops)*, volume 215 of *Lecture Notes in Informatics*, pages 123–138. Gesellschaft für Informatik, 2013. (Cited on pages 596 and 605.)

Daniel Grahl. *Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java*. PhD thesis, Karlsruhe Institute of Technology, 29 October 2015. (Cited on pages x, 351, 593 and 596.)

Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, PN87614, Tandem Computers, June 1985. (Cited on page 384.)

Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT — exploring runtime for .NET architecture and applications. *Electronic Notes in Theoretical Computer Science*, 144(3):3–26, 2006. Proceedings of the Workshop on Software Model Checking (SoftMC 2005), Software Model Checking, Edinburgh, UK, 2005. (Cited on page 384.)

John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993. (Cited on page 194.)

Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000a. (Cited on page 108.)

Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000b. (Cited on page 108.)

Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005. (Cited on page 280.)

Reiner Hähnle and Richard Bubel. A Hoare-style calculus with explicit state updates. In Zoltán Instenes, editor, *Proc. Formal Methods in Computer Science Education (FORMED)*, Electronic Notes in Theoretical Computer Science, pages 49–60. Elsevier, 2008. (Cited on pages x, 7, 15 and 572.)

Reiner Hähnle, Wolfram Menzel, and Peter Schmitt. Integrierter deduktiver Software-Entwurf. *Künstliche Intelligenz*, pages 40–41, December 1998. (Cited on page 1.)

Reiner Hähnle, Markus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In Jamie Andrews and Elisabetta Di Nitto, editors, *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium*, pages 143–146. ACM Press, 2010. (Cited on pages 8, 384 and 412.)

Reiner Hähnle, Nathan Wasser, and Richard Bubel. Array abstraction with symbolic pivots. In Erika Ábrahám, Marcello Bonsangue, and Broch Einar Johnsen, editors, *Theory and Practice*

*of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 104–121. Springer, 2016. (Cited on pages 184 and 187.)

Christian Hammer. *Information Flow Control for Java – A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), July 2009. (Cited on pages 596 and 605.)

Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages 87–96. IEEE, March 2006. (Cited on page 454.)

David Harel. *First-Order Dynamic Logic*. Springer, 1979. (Cited on page 65.)

David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984. (Cited on page 49.)

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000. (Cited on pages 12, 49 and 330.)

Trevor Harmon and Raymond Klefstad. A survey of worst-case execution time analysis for real-time Java. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, Long Beach, California, USA*, pages 1–8. IEEE Press, 2007. (Cited on page 582.)

Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In Katharina Morik, editor, *GWAI-87, 11th German Workshop on Artificial Intelligence, Geseke, 1987, Proceedings*, volume 152 of *Informatik Fachberichte*, pages 201–210. Springer, 1987. (Cited on page 12.)

Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic execution debugger (SED). In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification, 14th International Conference, RV, Toronto, Canada*, volume 8734 of *LNCS*, pages 255–262. Springer, 2014a. (Cited on pages x, 8 and 384.)

Martin Hentschel, Reiner Hähnle, and Richard Bubel. Visualizing unbounded symbolic execution. In Martina Seidl and Nikolai Tillmann, editors, *Proceedings of Testing and Proofs (TAP) 2014*, LNCS, pages 82–98. Springer, July 2014b. (Cited on pages x and 386.)

Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In Emil Sekerinski Elvira Albert and Gianluigi Zavattaro, editors, *Proceedings of the 11th International Conference on Integrated Formal Methods*, volume 8739 of *LNCS*, pages 55–70. Springer, 2014c. (Cited on pages x and 566.)

Martin Hentschel, Reiner Hähnle, and Richard Bubel. Can formal methods improve the efficiency of code reviews? In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods, 12th International Conference, IFM, Reykjavik, Iceland*, volume 9681 of *LNCS*, pages 3–19. Springer, 2016. (Cited on pages 8 and 18.)

Mihai Herda. Generating bounded counterexamples for KeY proof obligations. Master thesis, Karlsruhe Institute of Technology, January 2014. (Cited on page 439.)

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10):576–580, 583, October 1969. (Cited on pages 7, 208, 234, 349, 571 and 574.)

C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, Berlin, Heidelberg, 1971. (Cited on page 299.)

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. (Cited on page 302.)

C.A.R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, Revised Selected Papers and Discussions*, volume 4171 of *LNCS*, pages 1–18. Springer, 2005. (Cited on page 289.)

Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003. (Cited on pages 6 and 7.)

Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamaric. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In Mauro Pezzè and Mark Harman, editors,

*International Symposium on Software Testing and Analysis, ISSTA, Lugano, Switzerland*, pages 268–279. ACM, 2013. (Cited on page 18.)

Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in Java Card. In Michel Wermelinger and Tiziana Margaria, editors, *Proc. Fundamental Approaches to Software Engineering (FASE), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004. (Cited on page 377.)

Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France*, 2006. (Cited on page 374.)

Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *14th International Symposium on Parallel and Distributed Computing (ISPDC 2015)*, pages 165–174. IEEE Computer Society, 2015. (Cited on pages 378 and 380.)

Marieke Huisman, Wolfgang Ahrendt, Daniel Bruns, and Martin Hentschel. Formal specification with JML. Technical Report 2014-10, Department of Informatics, Karlsruhe Institute of Technology, 2014. (Cited on page 193.)

Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012. *International Journal on Software Tools for Technology Transfer*, 17(6):647–657, 2015. (Cited on page 289.)

James J. Hunt, Fridtjof B. Siebert, Peter H. Schmitt, and Isabel Tonin. Provably correct loops bounds for realtime Java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM. (Cited on page 583.)

Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004. (Cited on page 572.)

Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1–2):65–98, 2014. (Cited on page 18.)

ISO. ISO 26262, road vehicles – functional safety. published by the International Organization for Standardization, 2011. (Cited on page 424.)

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions Software Engineering and Methodology*, 11(2):256–290, April 2002. (Cited on page 438.)

Daniel Jackson. Alloy: A logical modelling language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland. Proceedings*, volume 2651 of *LNCS*, page 1. Springer, 2003. (Cited on page 240.)

Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008. (Cited on pages 2 and 384.)

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA*, pages 271–282. ACM, 2011. (Cited on page 241.)

Bart Jacobs and Erik Poll. A logic for the Java Modeling Language. In Heinrich Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering, 4th International Conference (FASE), Genova, Italy*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001. (Cited on pages 195 and 243.)

Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997. (Cited on page 252.)

Bart Jacobs, Hans Meijer, and Erik Poll. VerifiCard: A European project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001. (Cited on page 353.)

Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 202–219. Springer, 2003. (Cited on page 88.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, and Frank Piessens. The VeriFast program verifier – a tutorial for Java Card developers. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, September 2011a. (Cited on page 377.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA. Proceedings*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011b. (Cited on pages 377 and 378.)

Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules. In Michael Butler and Wolfram Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, pages 402–416. Springer, June 2011c. (Cited on page 350.)

JavaCardRTE. *Java Card 3 Platform Runtime Environment Specification, Classic Edition, Version 3.0.4, Oracle*, September 2012. (Cited on pages 5, 353 and 354.)

JavaCardVM. *Java Card 3 Platform Virtual Machine Specification, Classic Edition, Version 3.0.4, Oracle*, September 2012. (Cited on page 354.)

Trevor Jennings. SPARK: the libre language and toolset for high-assurance software engineering. In Greg Gicca and Jeff Boleng, editors, *Proceedigngs, Annual ACM SIGAda International Conference on Ada, Saint Petersburg, Florida, USA*, pages 9–10. ACM, 2009. (Cited on page 5.)

Ran Ji. *Sound programm transformation based on symbolic execution and deduction*. PhD thesis, Darmstadt University of Technology, Department of Computer Science, 2014. (Cited on pages x, 482, 491 and 492.)

Ran Ji and Reiner Hähnle. Information flow analysis based on program simplification. Technical Report TUD-CS-2014-0877, Department of Computer Science, 2014. (Cited on page 492.)

Ran Ji, Reiner Hähnle, and Richard Bubel. Program transformation based on symbolic execution and deduction. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods: 11th International Conference, SEFM 2013, Madrid, Spain*, volume 8137 of *LNCS*, pages 289–304. Springer, 2013. (Cited on pages x and 5.)

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proceedigns, 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011. (Cited on page 6.)

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. (Cited on page 351.)

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993. (Cited on page 475.)

Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ilkka Niemelä. LCT: An open source concolic testing tool for Java programs. In Pierre Ganty and Mark Marron, editors, *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011)*, pages 75–80, 2011. (Cited on page 450.)

Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976. (Cited on page 234.)

Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada. Proceedings*, volume 4085 of *LNCS*, pages 268–283, Berlin, 2006. Springer. (Cited on pages 13, 320 and 322.)

Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects Computing*, 23(3):267–288, 2011. (Cited on pages ix, 241, 290, 320 and 322.)

Shmuel Katz and Zohar Manna. Towards automatic debugging of programs. *ACM SIGPLAN Notices*, 10(6):143–155, 1975. Proceedings of the International Conference on Reliable software, Los Angeles. 1975. (Cited on page 383.)

Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR. In *8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2016. To appear. (Cited on page 483.)

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, July 1976. (Cited on pages 4, 67 and 383.)

Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Proceedings of Trustworthy Global Computing (TGC)*, volume 4661 of *LNCS*, pages 244–262. Springer, 2006. (Cited on page 606.)

Laurie Kirby and Jeff Paris. Accessible independence results for Peano Arithmetic. *Bulletin of the London Mathematical Society*, 14(4), 1982. (Cited on page 40.)

Michael Kirsten. Proving well-definedness of JML specifications with KeY. Studienarbeit, KIT, 2013. (Cited on pages 254 and 287.)

Vladimir Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538:124–139, 2014. (Cited on page 470.)

Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland. Proceedings*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011. (Cited on pages 18 and 289.)

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. (Cited on page 9.)

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison–Wesley, third edition, 1998. (Cited on page 558.)

Dexter Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990. (Cited on page 49.)

Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008. (Cited on page 537.)

Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings, 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014. (Cited on page 97.)

Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, Berkeley, California, USA*, pages 538–553, Oakland, California, USA, 2011. IEEE Computer Society. (Cited on pages 593, 594, 595 and 605.)

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving noninterference of Java programs. In Cédric Fournet and Michael Hicks, editors, *28th IEEE Computer Security Foundations Symposium*, pages 305–319. IEEE Computer Society, 2015. (Cited on pages 596 and 605.)

Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland. (Cited on page 291.)

Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. (Cited on page 454.)

Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In Bernhard Beckert, editor, *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21 Bremen, Germany*, volume 259, pages 85–103. CEUR Workshop Proceedings, July 2007. (Cited on page 17.)

Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, Massachusetts Institute of Technology, December 1988. (Cited on pages 219, 260, 292 and 293.)

Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In Ursula Martin and Jeannette M. Wing, editors, *Proceedings of the First International Workshop on Larch, 1992*, Workshops in Computing, pages 159–184, New York, NY, 1993. Springer. (Cited on page 194.)

Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000. (Cited on pages 219 and 293.)

Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006. (Cited on pages 219 and 293.)

Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995. (Cited on page 293.)

Gary T. Leavens and Jeanette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10(1):59–75, 1998. (Cited on page 281.)

Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA. Proceedings*, pages 221–236, New York, NY, USA, 2006a. ACM. (Cited on page 289.)

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3): 1–38, 2006b. (Cited on pages 193 and 253.)

Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007. (Cited on page 289.)

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31, 2013. Draft Revision 2344. (Cited on pages ix, 2, 13, 193, 208, 233, 243, 244, 245, 247, 248, 253, 261, 262, 280, 322, 328, 621 and 628.)

Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008. (Cited on page 473.)

K. Rustan M. Leino. *Towards Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03. (Cited on page 289.)

K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada*, volume 33, pages 144–153. ACM, October 1998. (Cited on pages 320 and 347.)

K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6): 281–288, 2005. (Cited on page 76.)

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, 2010, Revised Selected Papers*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010. (Cited on pages 2, 7, 10, 241 and 348.)

K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, Edition 0. In Gary T. Leavens, Peter W. O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE, Edinburgh, UK*, Edinburgh, UK, 2010. (Cited on page 296.)

K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004. (Cited on page 215.)

K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130, New York, NY, March 2006. Springer. (Cited on page 350.)

K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal. Proceedings*, volume 1383 of *LNCS*, pages 302–305. Springer, 1998. (Cited on page 240.)

K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002. (Cited on pages 302 and 322.)

K. Rustan M. Leino, Greg Nelson, and J.B. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000. (Cited on pages 195 and 240.)

K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5), pages 246–257, New York, NY, June 2002. ACM. (Cited on page 348.)

K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009. (Cited on pages 350 and 378.)

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 42–54. ACM, 2006. (Cited on page 473.)

Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009. (Cited on page 473.)

Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, pages 17–34, May 1988. (Cited on pages 218, 219 and 292.)

Barbara Liskov and Jeanette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 16–28, Washington DC, USA, 1993. ACM Press. (Cited on pages 217 and 292.)

Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. (Cited on pages 218 and 292.)

Sarah M. Loos, David W. Renshaw, and André Platzer. Formal verification of distributed aircraft controllers. In Calin Belta and Franjo Ivancic, editors, *Proc. 16th Intl. Conference on Hybrid Systems: Computation and Control, HSCC, Philadelphia, PA, USA*, pages 125–130. ACM, 2013. (Cited on page 6.)

Claude Marché and Nicolas Rousset. Verification of Java Card applets behavior with respect to transactions and card tears. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), Pune, India*, pages 137–146. IEEE CS Press, 2006. (Cited on page 377.)

Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *J. Logic and Algebraic Programming*, 58:89–106, 2004. (Cited on pages 195, 239 and 353.)

John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62, Munich, Germany*, pages 21–28. North-Holland, 1962. (Cited on page 41.)

John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 19:33–41, 1967. Proceedings of Symposia in Applied Mathematics. 1967. (Cited on page 473.)

José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, Second*

*International Joint Conference, IJCAR 2004, Cork, Ireland, Proceedings*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004. (Cited on page 64.)

Bertrand Meyer. From structured programming to object-oriented design: The road to Eiffel. *Structured Programming*, 1:19–39, 1989. (Cited on page 246.)

Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992. (Cited on pages 13, 194, 289 and 291.)

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997. (Cited on pages 194 and 291.)

Alysson Milanez, Dênnis Sousa, Tiago Massoni, and Rohit Gheyi. JMLOK2: A tool for detecting and categorizing nonconformances. In Uirá Kulesza and Valter Camargo, editors, *Congresso Brasileiro de Software: Teoria e Prática*, pages 69–76, 2014. (Cited on page 239.)

Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–72, 1972. Proceedings of the 7th Annual Machine Intelligence Workshop, Edinburgh, 1972. (Cited on page 473.)

Andrzej Mostowski. On a generalization of quantifiers. *Fundamenta Mathematicæ*, 44(1):12–36, 1957. (Cited on page 248.)

Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE), Edinburgh, Proceedings*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005. (Cited on pages 354, 376 and 609.)

Wojciech Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings, Formal Methods (FM) 2006, Hamilton, Ontario, Canada*, volume 4085 of *LNCS*, pages 444–459. Springer, August 2006. (Cited on pages 354 and 376.)

Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Proceedings of 4th International Verification Workshop (VERIFY) in connection with CADE-21, Bremen, Germany, 2007*, 2007. (Cited on pages 3, 6, 354, 376 and 609.)

Wojciech Mostowski. Dynamic frames based verification method for concurrent Java programs. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE, San Francisco, CA, USA, Revised Selected Papers*, volume 9593 of *LNCS*, pages 124–141. Springer, 2015. (Cited on pages 3, 378 and 380.)

Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Application Conference CARDIS 2008*, volume 5189 of *LNCS*, pages 1–16. Springer, September 2008. (Cited on pages 354 and 361.)

Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. In *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA*, pages 109–116. ACM, 2015. (Cited on pages 311 and 316.)

Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. *Transactions Modularity and Composition*, 1:238–267, 2016. (Cited on page 311.)

Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, Berlin, 2002. (Cited on pages 296, 348 and 350.)

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, February 2003. (Cited on pages 233 and 348.)

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006. (Cited on page 215.)

Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Frank Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007. (Cited on page 16.)

Andrew C. Myers. JFlow: practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA*, pages 228–241, New York, NY, USA, 1999. ACM. (Cited on pages 454 and 606.)

Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, second edition, 2004. (Cited on page 576.)

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, Berkeley, California, USA*, pages 165–179, may 2011. (Cited on page 455.)

David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376 (3):205–224, 2007. (Cited on page 210.)

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. (Cited on pages 2, 10 and 108.)

Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May / June 1997. (Cited on page 230.)

Kirsten Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*, ACM monograph series. Academic Press, 1981. (Cited on page 291.)

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France. Proceedings*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. (Cited on pages 241 and 349.)

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 268–280. ACM, January 2004. (Cited on page 241.)

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Transactions on Programming Languages and Systems*, 31(3):11:1–11:50, April 2009. (Cited on page 349.)

Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, 1996, Proceedings*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996. (Cited on page 108.)

Pierre Le Pallec, Ahmad Saif, Olivier Briot, Michael Bensimon, Jérome Devisme, and Marilyne Eznack. NFC cardlet development guidelines v2.2. Technical report, Association Française du Sans Contact Mobile, 2012. (Cited on pages 354, 355 and 360.)

Matthew Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership (IWACO)*, volume 23. ACM, 2007. position paper. (Cited on page 349.)

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Notices*, 40(1): 247–258, January 2005. (Cited on pages 349 and 350.)

Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013. (Cited on page 449.)

Christine Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 45–95. Springer, 2012. (Cited on page 10.)

Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. How test generation helps software specification and deductive verification in Frama-C. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK. Proceedings*, LNCS, pages 204–211. Springer, 2014. (Cited on page 449.)

André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004. (Cited on page 65.)

André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010. (Cited on page x.)

André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning,*

*4th International Joint Conference, IJCAR, Sydney, Australia*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008. (Cited on pages 6 and 16.)

Arndt Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technical University of Munich, 1997. Habilitation thesis. (Cited on page 215.)

Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods – 19th International Symposium, Singapore. Proceedings*, volume 8442 of *LNCS*, pages 514–530. Springer, 2014. (Cited on page 349.)

Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015: Formal Methods - 20th Intl. Symp., Oslo, Norway*, volume 9109 of *LNCS*, pages 414–434. Springer, 2015. (Cited on page 3.)

Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, Montreal, Quebec, Canada*, pages 535–552. ACM, 2007. (Cited on page 383.)

Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual IEEE Symposium on Foundation of Computer Science, Houston, TX, USA. Proceedings*, pages 109–121. IEEE Computer Society, 1977. (Cited on pages 12 and 49.)

Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005. (Cited on pages 206 and 255.)

Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. Modularizing crosscutting contracts with AspectJML. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland. Proceedings*, pages 21–24, New York, NY, USA, 2014. ACM. (Cited on page 239.)

John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Foundations of Computing, pages 13–24. The MIT Press, 1994. Reprint of the original 1975 paper. (Cited on page 252.)

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 349 and 378.)

Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal on Software Tools for Technology Transfer, STTT*, 8(3):280–299, 2006. (Cited on page 239.)

Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Decision procedures for region logic. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA. Proceedings*, volume 7148 of *LNCS*, pages 379–395, Berlin Heidelberg, 2012. Springer. (Cited on page 350.)

Andreas Roth. *Specification and Verification of Object-oriented Software Components*. PhD thesis, Universität Karlsruhe, 2006. (Cited on page 296.)

RTCA. DO-178C, Software considerations in airborne systems and equipment certification. published as RTCA SC-205 and EUROCAE WG-12, 2012. (Cited on page 424.)

James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, Reading/MA, 2nd edition, 2010. (Cited on page 240.)

Philipp Rümmer. Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2006. (Cited on pages 576 and 579.)

Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. Karlsruhe, KIT, Diss., 2014. (Cited on pages 454, 455, 456, 457, 458, 460, 463, 467, 593 and 595.)

Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software International Conference, Turin, FoVeOOS 2011, Revised Selected Papers*, volume 7421 of *LNCS*, pages 232–249. Springer, 2012. (Cited on pages 455 and 458.)

Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore. Proceedings*, volume 8442 of *LNCS*, pages 579–594. Springer, 2014. (Cited on pages 455 and 462.)

Steffen Schlager. Handling of integer arithmetic in the verification of Java programs. Diplomarbeit, University of Karlsruhe, July 10 2002. (Cited on pages 230 and 245.)

Peter H. Schmitt. A computer-assisted proof of the Bellman-Ford lemma. Technical Report 2011,15, Karlsruhe Institute of Technology, Fakultät für Informatik, 2011. (Cited on page 280.)

Peter H. Schmitt and Mattias Ulbrich. Axiomatization of typed first-order logic. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway. Proceedings*, volume 9109 of *LNCS*, pages 470–486. Springer, 2015. (Cited on page 47.)

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*, pages 138–152. Springer, 2010. (Cited on page ix.)

Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25:452–499, 2003. (Cited on page 491.)

Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary. Proceedings*, volume 4961 of *LNCS*, pages 261–275, Berlin, April 2008. Springer. (Cited on page 348.)

Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst*, 34(1):2, 2012. (Cited on pages 350 and 378.)

Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015. (Cited on pages 2 and 18.)

J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992. (Cited on page 240.)

Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Institut für Informatik, Universität Augsburg, Germany, July 2005. (Cited on page 239.)

Jacques Stern. Why provable security matters? In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland. Proceedings*, volume 2656 of *LNCS*, pages 449–461. Springer, 2003. (Cited on page 607.)

Christian Sternagel. Proof pearl — A mechanized proof of GHC's mergesort. *Journal of Automated Reasoning*, pages 357–370, 2013. (Cited on page 609.)

Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, (IWACO) at ECOOP 2008, Paphos, Cyprus*, pages 1–9. ACM, 2009. (Cited on page 350.)

Robert D. Tennent. *Specifying Software: a Hands-On Introduction*. Cambridge University Press, 2002. (Cited on page 572.)

Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. (Cited on page 450.)

Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Michel Wermelinger and Harald Gall, editors, *Proc. 10th European Software Engineering Conference/13th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, 2005, Lisbon, Portugal*, pages 253–262. ACM Press, 2005. (Cited on page 4.)

Kerry Trentelman. Proving correctness of Java Card DL taclets using Bali. In Bernhard Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany*, pages 160–169, 2005. (Cited on page 64.)

Thomas Tuerk. A formalisation of smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany. Proceedings*, volume 5674 of *LNCS*, pages 469–484. Springer, 2009. (Cited on page 241.)

Mattias Ulbrich. A dynamic logic for unstructured programs with embedded assertions. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*, pages 168–182. Springer, 2011. (Cited on page 473.)

Mattias Ulbrich. *Dynamic Logic for an Intermediate Language. Verification, Interaction and Refinement*. PhD thesis, Karlsruhe Institut für Technologie, KIT, 2013. (Cited on pages 36 and 473.)

Bart van Delft and Richard Bubel. Dependency-based information flow analysis with declassification in a program logic. *Computing Research Repository (CoRR)*, 2015. (Cited on page 471.)

Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Genova, Italy*, volume 2031 of *LNCS*, pages 299–312, 2001. (Cited on page 195.)

Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the Z notation. In *25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, Chicago, IL, USA*, pages 351–356. IEEE Computer Society, 2001. (Cited on pages 424 and 425.)

David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. (Cited on page 64.)

Simon Wacker. Blockverträge. Studienarbeit, Karlsruhe Institute of Technology, 2012. (Cited on pages 238, 466 and 623.)

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999. (Cited on pages 1, 13 and 240.)

Nathan Wasser. Generating specifications for recursive methods by abstracting program states. In Xuandong Li, Zhiming Liu, and Wang Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China. Proceedings*, pages 243–257. Springer, 2015. (Cited on page 189.)

Benjamin Weiß. Predicate abstraction in a program logic calculus. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany. Proceedings*, volume 5423 of *LNCS*, pages 136–150. Springer, 2009. (Cited on page 474.)

Benjamin Weiß. *Deductive Verification of Object-Oriented Software — Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe, January 2011. (Cited on pages ix, 241, 243, 251, 290, 306, 307, 319, 322, 335, 336, 338 and 341.)

Florian Widmann. Crossverification of while loop semantics. Diplomarbeit, Fakultät für Informatik, KIT, 2006. (Cited on page 101.)

Niklaus Wirth. Modula: a language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977. (Cited on page 291.)

Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer, STTT*, 14(5):567–588, 2012. (Cited on page 6.)

Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The certification of the mondex electronic purse to ITSEC level E6. *Formal Aspects of Computing*, 20(1):5–19, 2008. (Cited on page 605.)

Jooyong Yi, Robby, Xianghua Deng, and Abhik Roychoudhury. Past expression: encapsulating pre-states at post-conditions by means of AOP. In *Proceedings of the 12th annual international conference on Aspect-oriented software development, (AOSD), Fukuoka, Japan*, pages 133–144. ACM, 2013. (Cited on page 249.)

Lei Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, 2013, 2013. (Cited on page 3.)

Marina Zaharieva-Stojanovski and Marieke Huisman. Verifying class invariants in concurrent programs. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France. Proceedings*, volume 8411 of *LNCS*, pages 230–245. Springer, 2014. (Cited on page 216.)

Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Programming Language Design and Implementation (PLDI)*, pages 349–361, New York, NY, 2008. ACM. (Cited on page 296.)

Andreas Zeller. *Why programs fail—A guide to systematic debugging*. Elsevier, 2nd edition, 2006. (Cited on page 412.)

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. (Cited on page 423.)

Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The Next Generation. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*. Springer, 2010. (Cited on page 239.)