New Foundations for Separation Logic

Hiep, H.A.

Appendix A

Classical (higher-order) logic

In this chapter we recall the definitions and results of classical logic, upon which the rest of this thesis depends (see also [135, 217, 213, 214, 203, 31, 152, 91, 9, 82]). Readers already familiar with classical logic may quickly skim this chapter: no new results are presented. However, this chapter is included for the purpose of completeness.[1]

Logic is used as a formal description language. With such language, we give descriptions with an intention to *describe* or *assert* what is the case in a universe. Logic has versatile uses in computer science. For example, logic can be used to describe the universe as seen from the perspective of a computer. To be more specific, we can use logic to describe the possible memory states of a computer at a particular instant in time. Primitive descriptions can fix what values are held in certain places of memory, or describe relations between the values such places of memory hold. Complex descriptions are constructed by composing simpler descriptions, e.g. by conditionals between descriptions or quantification over possible values.

There are different orders of languages. In a zeroth-order language one can speak of primitive propositions and their logical connection. In a first-order language one furthermore has the ability to speak about, and quantify over, *individuals*, which are the elements in a universe, or domain of discourse. This distinguishes a first-order language from a zeroth-order language, since in the latter one cannot quantify over individuals. Sometimes first-order logic is called predicate logic, whereas zeroth-order logic is called propositional logic. In a second-order language one goes beyond the ability to quantify over individuals, and one can also quantify over properties of individuals (see also [216]). In a third-order language, one can quantify over properties of properties of individuals, and so on for higher-order languages.

[1] After submitting some parts of this thesis to a conference on logic in computer science, one of the peer-reviewers made a claim that was directly in contradiction with a well-known result, also included in this chapter: Gödel's completeness theorem. Thus, without recalling the completeness theorem, this thesis would not be complete.

We consider logic from three perspectives. In the first perspective, the *syntactic perspective*, one looks at the formal structure of the description language: how to form sequences of symbols called formulas, and systems to symbolically transform formulas and to derive formulas from other formulas. In the second perspective, the *semantic perspective*, one is interested in the meaning of formulas by interpretation of the symbols and how symbolic transformations and deductions preserve meaning. In the last perspective, the *perspective of significance*, one is interested in how the syntactic and semantic perspectives are related. In some sense, the syntax and semantics of a language can be chosen freely, and making such choices are called design choices. To motivate some of these design choices, it is the relation between the syntactic and semantic perspectives which bears significance.

The significance of logic is that it is a language that is useful for describing the universe and to reason about such descriptions. We distinguish two levels: the level in which we speak about the universe using logic, and the level in which we speak about the logic itself. The first level is the *object-level*, in which elements of the universe, their transformations and interrelations, are the prime subject. On this level, formulas are used to describe properties about elements of the universe, and we are interested in the logical connections between different properties (of the elements of the universe) that formulas can describe. The significance, or usefulness, of a logic depends on the richness of the object-level, in what properties can and can not be expressed by formulas. The second level is the *meta-level*, in which we study the properties of the syntax and semantics themselves. These properties are called meta-properties, to distinguish them from the properties which are described by formulas on the object-level. In particular, first-order logic is a well-known logic with very rich meta-properties. An example meta-property is the relationship between the so-called syntactic consequence relation between sets of formulas and the semantic consequence relation between sets of formulas.

There are many different logics described in the scientific and philosophic literature, including: classical logic, modal logic, intuitionistic logic. In this chapter we keep ourselves to classical logic: the logic in which the *law of the excluded middle* holds generally. This logic is most familiar to mathematicians and computer scientists.

This chapter proceeds as follows. Section A.1 presents the languages of logic, mostly from a syntactical perspective. Section A.2 introduces structures in which to interpret formulas, so to speak, and thus takes a semantic perspective. This section also introduces the concepts of validity and entailment. In Section A.3, going back to a syntactical perspective, we then introduce a proof system for deducing formulas. In Section A.5, we introduce terms as a shorthand for particular formulas and contexts. Section A.4 demonstrates the significance of logic by recalling important meta-properties that relates the semantic concept of entailment to the syntactic concept of deduction.

The ideas presented in this chapter are largely based on material as presented by any competent book on mathematical logic, such as [79, 31]. An introduction to higher-order logic can be found in [82]. The model theory is based on [47, 119]. The proof theory is based on [212, 26]. See also [17, 199, 83].

A.1 Assertion language

In this section we introduce the languages of logic. In later chapters we also speak of programming languages and program logics, so to avoid ambiguity, we may also speak of *assertion languages* to mean the languages introduced here.

We shall introduce not a single language, but a family of languages that is parameterized by *variables* and a *signature*. A signature consists of non-logical symbols which are taken as primitive, out of which a particular language is constructed. A language consists of *formulas*, or synonymously *assertions*, which are certain (finite) sequences of symbols. The formulas are formed using syntactic rules that depend on the signature one chooses.

Moreover, we restrict ourselves to recursive languages, meaning that for each language there must exist an algorithm that can decide whether a given sequence of symbols is a formula or not. Phrased differently, for each language we could systematically generate all sequences of symbols that are formulas and we could systematically generate all sequences of symbols that are non-formulas. This restriction is useful for computer scientists who want to implement such languages. We shall properly define the set of formulas below, and in such a way that the set is recursive. Before doing so, we introduce the concepts of arity and variables.

An *arity* is a (finite) sequence of arities. An arity is typically associated to a symbol to represent how many arguments, and the arity of each argument, are expected to be following that symbol. We write $(\alpha_1, \ldots, \alpha_n)$ for a sequence of $n > 0$ arities, where each of $\alpha_1, \ldots, \alpha_n$ are again arities. We write $()$ for the empty sequence. We say nullary to mean an arity of length 0, unary to mean an arity of length 1, binary to mean an arity of length 2, and so on. Note that, if we ignore all commas, the set of arities is the full Dyck language consisting of all strings of balanced parentheses.

The order of an arity is its maximal nesting depth, starting from first-order. We have that $()$ is a first-order arity since no arity is nested, $(())$ is a second-order arity since it has a first-order arity nested, $((),())$ is also a second-order arity since all nested arities are first-order, and $(((),()),())$ is a third-order arity since it has a nested second-order arity, and so on. We may also treat a natural number n as an arity, which has precisely n directly nested first-order arities $()$. As a special case, 0 is the arity $()$ since it contains no other nested arities. Arity 0 is first-order, whereas arity n for $n > 0$ is second-order. For example, $1 = (())$ and $2 = ((),())$. All second-order arities are a natural number arity n for some $n > 0$. We may mix parenthesis and natural numbers, for example $((),()) = (0,0)$.

Variables (or, more precisely, variable symbols) can be understood as names or as value placeholder symbols. Given two variables, we are able to recognize whether they are (syntactically) identical or not. Each variable has an arity associated to it. Two variables of different arity are necessarily different.

Definition A.1.1 (Variables)**.** There is a recursive set V of *variables*, such that each variable is associated to an arity, and for each arity there are infinitely many variables associated to that arity.

The order of a variable is the order of the arity of the variable. We write v^α

to mean that v is a variable with as associated arity α. Note that 'v' itself is not a variable, but a meta-variable standing for a variable symbol. We also use w^α, where 'w' is again a meta-variable standing for variable symbols, and we may use subscripts to obtain any number of such variables. The variables of arity () are called the *first-order* variables. These are also called the *individual variables*, and are typically denoted x, y, z (without superscript). The set of first-order variables is also denoted by V_1. Note that there are no zeroth-order variables, since in a zeroth-order language there is no need for variables. Variables that have a second-order arity are called *second-order* variables. These are also called *predicate variables* (if its arity is 1) or *relation variables* (if its arity is greater than 1) typically denoted P, Q, R. For example, Q^2 (or, equally, $Q^{(0,0)}$) is a binary relation variable (with arity 2). The set of second-order variables is denoted by V_2, and so on for higher-order variables. We may leave out arity superscripts if the arity of a variable is clear from context; otherwise, we leave the superscript in place.

Intuitively, variables are called that way since their meaning depends on the context and thus vary. In contrast, one may think of a signature as consisting of constant symbols. These symbols are called constant since their meaning does not depend on the context and remains fixed within one language. Variable symbols and constant symbols are separate, and both together are the *non-logical symbols*.

Definition A.1.2 (Signatures)**.** A *signature* is a recursive set of *constant symbols* such that each constant symbol is associated to a non-zero arity.

We typically denote a signature by Σ. The signatures as defined above are signatures *without parameters*. The order of a constant symbol is the order of its arity. We speak of constants to mean constant symbols of a signature.

The order of a signature is one less than the maximum order of its constants. A first-order signature thus has constants with at most second-order arity. In other words, a first-order signature has no constants with a third or higher-order arity. A second-order signature has constants with at most third-order arity. And so on for higher-order signatures.

There are no first-order constant symbols in any signature, since the only first-order arity is zero. The constant symbols of (second-order) arity 1 are called *predicate symbols*. The constant symbols of (second-order) arity n where $n > 1$ are called n-ary *relation symbols*.

Constant symbols of a signature are typically denoted C^α (with arity α, which may be dropped under the same proviso that holds for variable symbols), but specific signatures may also introduce additional notational conventions. For first-order signatures we may also use P, Q, R as constant symbols (but care must be taken not to confuse constant symbols and variable symbols).

Remark A.1.3. In some texts about logic, signatures also include '[individual] constant symbols' and 'function symbols' that are separate from predicate and relation symbols and used to build complex terms. We do not yet (need to) introduce terms, individual symbols and function symbols at this point, but we shall introduce them later in Section A.5. In here, we consider all symbols of a signature to be constant symbols, in the sense that their meaning does not depend on the context and remains fixed within one language.

For the remainder of this section, we fix a particular signature Σ. We now continue to define the fundamental concept of logic: formulas. More precisely, we define Σ-formulas, since their formulation depends on the chosen signature Σ. We may speak of formulas instead of Σ-formulas, if Σ is clear from context. In the sequel, ϕ and ψ are meta-variables standing for arbitrary formulas.

Definition A.1.4 (Formulas)**.** A *formula* is constructed inductively as follows:

1. $\bot$ is a formula,
2. $(x \doteq y)$ is a formula if x and y are individual variable symbols,
3. $\Xi(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})$ is a formula if Ξ is a non-logical symbol of arity $(\alpha_1, \ldots, \alpha_n)$ and $v_1^{\alpha_1}, \ldots, v_n^{\alpha_n}$ are variable symbols of the corresponding arity,
4. $(\phi \to \psi)$ is a formula if ϕ and ψ are formulas,
5. $(\forall v^\alpha \phi)$ is a formula if ϕ is a formula and v a variable symbol with arity α.

All formulas are constructed by one of these five clauses. Alternatively, we can define formulas by the following abstract grammar:

$$\phi, \psi ::= \bot \mid (x \doteq y) \mid \Xi(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n}) \mid (\phi \to \psi) \mid (\forall v^\alpha \phi)$$

The first three clauses construct primitive formulas, the last two clauses construct complex formulas. Note that in the third clause, variables with non-zero arity can be used in the place of the non-logical symbol. Parentheses, comma, $\bot$, $\doteq$, $\to$ and $\forall$ are logical symbols (and thus not part of the signature nor used as variables). The symbol $\doteq$ is called *identity*. We put the dot on the equality sign to distinguish (object-level) identity from (meta-level) equality. A formula can thus be seen as a finite sequence of logical and non-logical symbols.

We speak of the logical symbols in the following manner. Of the primitive formulas, $\bot$ is called *false*, $(x \doteq y)$ is called *identity* (as in 'x and y are identical'). If in $\Xi(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})$ we have that Ξ is a variable, then we speak of an *application*. Of the complex formulas, $(\phi \to \psi)$ is called (logical) *implication*, and $(\forall v^\alpha \phi)$ is called *universal quantification*.

Often when proving meta-properties of formulas, we proceed by induction on the *complexity* of formulas. There are different measures of complexity. Typically, we use as complexity measure the *height of a formula*, by viewing the formula as a parse tree and taking the height of that tree. Alternatively, one could take as complexity the *length of a formula*, by viewing a formula as a sequence of symbols and taking the length of that sequence.

We further have the logical symbols $\top$, $\neg$, $\not\doteq$, $\wedge$, $\vee$, $\leftrightarrow$, $\exists$ to construct formulas that are commonly used in classical logic. These logical symbols are given as abbreviations. Sometimes, we write **true** instead of $\top$, and **false** instead of $\bot$.

Definition A.1.5 (Logical abbreviations).

$$\begin{aligned}
\top &\text{ abbreviates } (\bot \to \bot)\\
(\neg\phi) &\text{ abbreviates } (\phi \to \bot)\\
(x \neq y) &\text{ abbreviates } (\neg(x \doteq y))\\
(\phi \wedge \psi) &\text{ abbreviates } (\neg(\phi \to (\neg\psi)))\\
(\phi \vee \psi) &\text{ abbreviates } (\neg((\neg\phi) \wedge (\neg\psi)))\\
(\phi \leftrightarrow \psi) &\text{ abbreviates } ((\phi \to \psi) \wedge (\psi \to \phi))\\
(\exists v^\alpha \phi) &\text{ abbreviates } (\neg(\forall v^\alpha(\neg\phi)))
\end{aligned}$$

The logical symbols $\wedge$, $\vee$, $\to$, $\leftrightarrow$ are called (logical) connectives, since they describe a connection between two formulas. Formulas of the form $(\phi \wedge \psi)$ are called *conjunctions*, and $(\phi \vee \psi)$ are called *disjunctions*, and $(\phi \leftrightarrow \psi)$ are called *bi-implications*. The logical symbols $\forall$ and $\exists$ are called (logical) quantifiers. We have two kinds of quantifiers: $\forall$ quantifies *universally*, and $\exists$ quantifies *existentially*. Note that some of the above syntactic abbreviations are arbitrary: it is also possible to introduce these logical connectives in alternative but equivalent ways.

To reduce the use of parentheses, we employ syntactical conventions for resolving ambiguity in case parentheses are dropped. The precedence of logical symbols, from strongest to weakest binding force, is: $\neg$, $\forall$, $\exists$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$. All connectives associate to the right. The process of adding back parentheses is called disambiguation. For example, $\forall x P(x) \wedge Q(x) \wedge P(x)$ disambiguates to $((\forall x P(x)) \wedge (Q(x) \wedge P(x)))$.

To further reduce the use of parentheses, we may employ a single dot after the variable that immediately follows a quantifier, that disambiguates into a pair of parentheses of which the closing parenthesis is placed as far as possible to the right without interfering with the parenthesis already present in the surrounding context. For example, $\forall x.\ P(x) \wedge Q(x)$ disambiguates to $\forall x(P(x) \wedge Q(x))$, and $(\forall x.\ P(x)) \wedge Q(x)$ disambiguates to $(\forall x(P(x))) \wedge Q(x)$.

We also have the syntactic convention that, given directly nested quantifiers of the same kind, such sequence of quantified variables may be listed as a (non-empty) sequence directly after the quantifier symbols. For example $\forall v_1^{\alpha_1}, v_2^{\alpha_2}, \ldots, v_n^{\alpha_n} \phi$ expands to $(\forall v_1^{\alpha_1}(\forall v_2^{\alpha_2}(\ldots(\forall v_n^{\alpha_n}\phi)\ldots)))$.

How are variable occurrences bound to quantifiers? The scope of a quantifier in the formula $(\forall v^\alpha \phi)$ is the variable v^α and the formula ϕ that follows it, and we say that v^α is bound to that quantifier. Informally, we can imagine the parse tree of how a formula is constructed. The possible leaves are variables with an associated arity. An occurrence of a symbol in a formula is a path in the parse tree leading to that symbol. A variable occurrence is an occurrence of a variable in a given formula. A variable occurs bound if it occurs as the immediate left child of a quantifier symbol, e.g. the formula $(\forall v^\alpha P(v^\alpha))$ has $\forall$ as root and v^α as left child and P as right child with v^α nested under it. A variable v^α occurs free if, following the occurrence of v^α as a path, from the root of the parse tree towards the leaf we do not encounter any quantifier with v^α as immediate left child. A variable occurrence of v^α falls under the scope of a quantifier if either it immediately follows a quantifier, or if we follow the path back to the root we encounter a quantifier binding the same variable. In other words, a variable occurs free if each of that variable occurrence does not fall under the scope of a quantifier.

We now define the set of free variables of a formula, being the set of all variables that occur free in it. Similarly, we define the set of bound variables of a formula.

Definition A.1.6 (Free and bound variables)**.** Given a formula ϕ. We define both the set of free variables $FV(\phi)$ and the set of bound variables $BV(\phi)$ inductively on the structure of ϕ as follows:

- $FV(\bot) = \emptyset = BV(\bot)$,
- $FV(x \doteq y) = \{x, y\}$ and $BV(x \doteq y) = \emptyset$,
- $FV(C(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})) = \{v_1^{\alpha_1}, \ldots, v_n^{\alpha_n}\}$ and $BV(C(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})) = \emptyset$,
- $FV(w^\alpha(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})) = \{w^\alpha, v_1^{\alpha_1}, \ldots, v_n^{\alpha_n}\}$ and $BV(w^\alpha(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})) = \emptyset$ where $\alpha = (\alpha_1, \ldots, \alpha_n)$,
- $FV(\phi \to \psi) = FV(\phi) \cup FV(\psi)$ and $BV(\phi \to \psi) = BV(\phi) \cup BV(\psi)$,
- $FV(\forall v^\alpha \phi) = FV(\phi) \setminus \{v^\alpha\}$ and $BV(\forall v^\alpha \phi) = BV(\phi) \cup \{v^\alpha\}$.

The set of variables $V(\phi)$ is defined $V(\phi) = FV(\phi) \cup BV(\phi)$.

Note that the second and third clause are discriminated by the non-logical symbol being either a constant symbol in our signature or a variable symbol. In the second clause, the constant symbol C must have arity $(\alpha_1, \ldots, \alpha_n)$, and in the third clause, the variable w must have arity $(\alpha_1, \ldots, \alpha_n)$: both constraints follow from the construction of formulas. Other concepts that are inductively defined on the structure of formulas follow a similar pattern.

Note that for every formula ϕ, we have that $V(\phi)$ is a finite set. This is easily seen, since every formula is a finite sequence of symbols, thus there can only be finitely many variables that occur in it. If $V(\phi) \subseteq \{v_1^{\alpha_1}, \ldots, v_n^{\alpha_n}\}$ we also write $\phi(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})$. For example, $\phi(x)$ is a formula ϕ in which at most x occurs free.

Definition A.1.7 (Sentences)**.** A *sentence* is a formula without free variables.

It is important to note that the context $\forall v^\alpha \ldots$ is so-called *referentially opaque* [205], meaning that the value each variable refers to in formulas under a quantifier may change. For example, $(x \doteq y)$ and $\forall x(x \doteq y)$ may have a different meaning: in one formula x could refer to a different value than the value x refers to in the other formula. This distinction is especially important when substitutivity comes in play, e.g. when replacing x by z, since referential opacity breaks our naïuve principle of 'substitution of equals for equals'.

Convention A.1.8 (Barendregt's variable convention). As a convention, we separate the names used for free variables and bound variables. Formally, a formula ϕ complies to this convention whenever it is the case that $FV(\phi) \cap BV(\phi) = \emptyset$.

It is possible to transform every formula into a formula that complies to the above convention. For that we introduce two concepts: fresh variables and variable renaming. Fresh variables are variables that do not occur in some context. A renaming allows us to transform a given formula into another formula in which variables are uniformly replaced. Now, by choosing appropriate fresh variables for bound variables, we can separate the free and bound variables of a given formula.

We can now motivate our choice in Definition A.1.1 to have for each arity an infinite supply of variables associated to that arity. This allows us to always be able, given a formula ϕ, to give a *fresh variable* of some arity. A variable v^α is fresh if it does not occur in a formula, i.e. $v^\alpha \notin V(\phi)$. For every variable that occurs bound in a formula, there are infinitely many fresh variables.

Definition A.1.9 (Variable renaming)**.** A *variable renaming* is a mapping π of variable symbols, such that variables of a given arity are mapped to variables of the same arity. We define the application $\pi(\phi)$ of a renaming π to a formula ϕ inductively on the structure of ϕ as follows:

- $\pi(\bot) = \bot$,
- $\pi(x \doteq y) = (\pi(x) \doteq \pi(y))$,
- $\pi(C(v_1^{\alpha_1}, \dots, v_n^{\alpha_n})) = C(\pi(v_1^{\alpha_1}), \dots, \pi(v_n^{\alpha_n}))$ for constant symbol C,
- $\pi(w^\alpha(v_1^{\alpha_1}, \dots, v_n^{\alpha_n})) = \pi(w^\alpha)(\pi(v_1^{\alpha_1}), \dots, \pi(v_n^{\alpha_n}))$ where $\alpha = (\alpha_1, \dots, \alpha_n)$,
- $\pi(\phi \to \psi) = \pi(\phi) \to \pi(\psi)$,
- $\pi(\forall v^\alpha \phi) = \forall \pi(v^\alpha)\pi(\phi)$.

It is worth pointing out that a renaming can potentially change multiple variables simultaneously. For example, $\exists y(\forall x P(x) \wedge Q(y))$ can be renamed to $\exists x(\forall y P(y) \wedge Q(x))$ by swapping x and y simultaneously. We may leave the exact mapping used to rename implicit. To explicitly denote a renaming, we use the notation $\binom{v_1^{\alpha_1} \, \dots \, v_n^{\alpha_n}}{w_1^{\alpha_1} \, \dots \, w_n^{\alpha_n}}$ to denote the renaming which simultaneously renames $v_1^{\alpha_1}$ into $w_1^{\alpha_1}$, …, $v_n^{\alpha_n}$ into $w_n^{\alpha_n}$ (from top to bottom) and leaves all other variables identical. In the previous example, the renaming $\binom{x \, y}{y \, x}$ is used.

When performing renaming, it is sometimes important that the variable that is renamed to is not captured by a quantifier. For example, in the formula $\forall x P(x, y) \wedge Q(y)$ we have that both occurrences of y have the same referents (they are both the same free variable), whereas if we rename y to x to obtain $\forall x P(x, x) \wedge Q(x)$, the one occurrence of variable x falls under the scope of the quantifier $\forall x$ whereas the other occurrence remains free. We say that v^{α_0} *remains free for* w^{α_1} *in* ϕ if all occurrences of w^{α_1} in ϕ do not fall under the scope of a quantifier binding v^{α_0}.

Note that for formulas that comply to Barendregt's convention, where the bound and free variables are separate, we do not have a problem with renaming of variables if the resulting formula also complies to Barendregt's convention. To ensure this, one could partition the variables into two sets: those potentially used for free variables, and those potentially used for bound variables. If a renaming retains the status of each variable (i.e. renaming free variables to other potentially free variables, and renaming bound variables to other potentially bound variables), no variable ever gets captured since free variables are never used under quantifiers. When one formula is obtained from another by the application of a renaming of the bound variables, we say the formulas are *alphabetic variants.*

Given non-empty lists of variables $\vec{v} = v_1^{\alpha_1}, \dots, v_n^{\alpha_n}$ and $\vec{w} = w_1^{\alpha_1}, \dots, w_n^{\alpha_n}$ such that the lists match up in length and arity. We write $\phi[\vec{v} := \vec{w}]$ to mean the operation where first ϕ is suitable renamed to avoid capture of the variables in $\vec{w}$, and then the renaming of the variables $\vec{v}$ into $\vec{w}$, respectively. This operation is also called (capture-avoiding) substitution.

As a convention, if we are given a formula $\phi(v_1^{\alpha_1}, \dots, v_n^{\alpha_n})$ with its free variables among the listed variables, then writing $\phi(w_1^{\alpha_1}, \dots, w_n^{\alpha_n})$ denotes a formula obtained from ϕ by substituting the variables $v_1^{\alpha_1}, \dots, v_n^{\alpha_n}$ by respectively $w_1^{\alpha_1}, \dots, w_n^{\alpha_n}$, leaving all other variables identical. For example, given $\phi(x, y)$, then $\phi(y, z)$ is the result obtained from simultaneously substitution x to y and y to z in ϕ. We need to take care to avoid variable capturing: if $\phi(x, y)$ is $\forall z(x \doteq y)$, then $\phi(y, z)$ must be $\forall w(y \doteq z)$ where we have renamed the bound variable z appropriately. We may sometimes be unclear, e.g. where $\phi(x)$ can have two meanings: either it declares that ϕ has the free variable x, or by $\phi(x)$ we mean ϕ with the identity renaming applied (which results in ϕ itself).

The order of a formula is determined as the maximum order of the arities of the variable symbols that occur in it, and a formula is called a *zeroth-order formula* if no variable occurs in it. So, if there is at least one variable occurrence and all variables that occur are first-order, then the formula in question is *first-order.* And so on for second and higher-order formulas.

In general, we can classify formulas by their order, and in doing so we also include the formulas of lower order. In a zeroth-order formula, no variables occur. In a first-order formula, all variables that occur are first-order and all constant symbols that occur have a second-order arity. Hence, the set of first-order formulas contains the set of zeroth-order formulas. In a second-order formula, all variables that occur are at most second-order and all constant symbols that occur are at most third-order. Hence, the set of second-order formulas contains the set of first-order formulas. And so on for higher-order formulas.

A *context* is a list of formulas, typically denoted $\phi_1, \dots, \phi_n$. We also have the empty list of formulas, for which we do not need any special notation. We may treat a single formula as a list of formulas of length 1 consisting of just that formula. If Γ and Δ are lists of formulas, then by Γ, Δ we mean the list formed by adjoining the formulas in the first list to the formulas in the second list. Consequently, by Γ, ϕ and ϕ, Γ we mean the lists formed by suffixing or prefixing the list consisting of a single formula ϕ to the list of formulas Γ, respectively.

On the one hand, contexts are syntactic and finitary objects: every formula is a finitary object, and every list of formulas can be seen as a finite sequence of formulas. On the other hand, we now introduce theories, which are possibly infinitary objects and in some cases are entirely semantic.

Definition A.1.10 (Theories)**.** A *theory* is a set of sentences.

A *first-order theory* is a set of first-order sentences, a *second-order* theory is a set of second-order sentences, and so on for higher orders. A *finite* theory is a theory where its set of sentences is finite, and an *enumerable* theory is a theory with a countable set of sentences. In particular, we shall look at three classes of theories: satisfiable theories, consistent theories, and complete theories.

In Section A.2, we introduce the semantics of formulas, and in particular we define when a theory is *satisfiable* (also called *semantically consistent*). A theory Γ is satisfiable (viz. semantically consistent) if there exists a structure for which all sentences in the theory are satisfied (one may think that the theory 'consists of' at least one such structure). In a satisfiable theory Γ, it must be the case that not all sentences are in Γ. However, the converse, if not all sentences are in Γ, does not necessarily imply that Γ is satisfiable.

Then, in Section A.3, we introduce syntactic proof systems and we define when a theory is *deductively closed*. A theory Γ is *deductively consistent* (or consistent in short) if it is not possible to deduce **false** from it. Equivalently, the theory Γ is consistent if there exists some sentence that is not contained in the deductive closure of Γ.

A theory Γ is *complete* if for every sentence ϕ, either $\phi \in \Gamma$ or $(\neg\phi) \in \Gamma$. It is called complete in the sense that it is no longer possible to add any more sentences without the theory turning inconsistent: if one adds a sentence to a complete theory that was not yet contained in it, closing the resulting theory deductively would result in an inconsistent theory.

The significance of these definitions is described in Section A.4, where we give the main result that the syntactic proof systems are, in a precise sense, adequate for our semantics: the notion of syntactic consistency coincides exactly with the notion of semantic consistency. Sometimes this is also called soundness and completeness (not to be confused with complete theories).

A.2 Basic model theory

In this section we introduce models of the languages we introduced above, in the style of Tarksi. Models are also called *structures*. Structures are used to give meaning to formulas of a language. We thus take a semantic perspective in this section. The meaning of formulas builds on two concepts: interpretations and valuations. Each structure fixes a *domain* of discourse, also called the universe. The domain restricts the values that are possible. Each structure also fixes the interpretation of constant symbols, assigning to each constant symbol a value. Structures further induce valuations, which assign to each variable symbol a value. Given both an interpretation (for the constant symbols) and valuation (for the variable symbols), we are able to give meaning to formulas relative to an ambient structure.

Both interpretations and valuations employ the concept of value, albeit assigning them to constants or variables, respectively. Values are structured by an arity and range over the domain. The values of first-order symbols range directly over the elements of our domain. There are no first-order constant symbols, only first-order variables. The values of second-order symbols range over particular sets comprising elements of our domain, depending on their arity. Moreover, the role of quantifiers in our language is to modify valuations, and thereby varying the value of variables depending on their context.

Definition A.2.1 (Values)**.** Given a domain D. Let $D[\alpha]$ denote the set of values of D of arity α. A *value* of D of arity α is constructed inductively:

1. An element $d \in D$ is a value of D of arity 0.

2. A set $S \subseteq D[\alpha_1] \times \ldots \times D[\alpha_n]$ is a value of D of arity $(\alpha_1, \ldots, \alpha_n)$.

All values are constructed by one of these two clauses.

It is easy to see that $D[0] = D$ and $D[(\alpha_1, \ldots, \alpha_n)] = \mathcal{P}(D[\alpha_1] \times \ldots \times D[\alpha_n])$ where $\mathcal{P}$ denotes the powerset operator on sets. Hence we have $D[1] = \mathcal{P}(D)$, $D[2] = \mathcal{P}(D \times D)$, and $D[n] = \mathcal{P}(D^n)$ for arities $n > 1$.

Example A.2.2. Let $\mathbb{N}$ be our domain. We then have the following values of $\mathbb{N}$:

- *First-order values:*

  $0, 1, 2, \ldots$ are in $\mathbb{N}[0] = \mathbb{N}$. Every natural number is a first-order value.

- *Second-order values:*

  $\{1, 3, 5, \ldots\}$ is in $\mathbb{N}[1] = \mathcal{P}(\mathbb{N})$, $\{(0,1),(1,2),(2,3),\ldots\}$ is in $\mathbb{N}[2] = \mathcal{P}(\mathbb{N} \times \mathbb{N})$, et cetera. Every possible set of natural numbers is a second-order value in $\mathbb{N}[1]$, and every possible binary relation on natural numbers is a second-order value in $\mathbb{N}[2]$, and so on.

- *Third-order values:*

  $\{(f, S) \mid f : \mathbb{N} \rightharpoonup \mathbb{N} \text{ and } S = \mathit{dom}(f)\}$ in $\mathbb{N}[(2,1)] = \mathcal{P}(\mathcal{P}(\mathbb{N} \times \mathbb{N}) \times \mathcal{P}(\mathbb{N}))$. This value describes a relational between a relation f and a set S, where $f : \mathbb{N} \rightharpoonup \mathbb{N}$ means that f is a partial function on $\mathbb{N}$, and $\mathit{dom}(f)$ is the set of elements on which f is defined. *End of Example.*

We have introduced values in this way to be able to attend to the higher-order aspect of languages, namely how to give a value to second and higher-order variables. We first introduce the so-called *standard* model theory of logic. In the standard model, variables range over all values. It is also possible to consider a different semantics of logic, resulting in the so-called *general* model theory of logic, where variables range over a restricted set of values.

As can be seen in the example above, the first-order values are elements of the domain. Sometimes we use *elementary* (or the adjective *elementarily*) as a synonym of first-order, to remind the reader of this fact.

For the remainder of this section, we again fix a particular signature Σ.

Definition A.2.3 (Structures)**.** A *structure* $\mathfrak{A}$ is a pair of a *domain* A (a set of elements) and an *interpretation* $\mathcal{I}$. An interpretation $\mathcal{I}$ assigns every constant symbol of arity α to a value of A of arity α.

Given that C is a constant symbol, we write $C^{\mathcal{I}}$ to mean the value given to C by the interpretation $\mathcal{I}$ (and it should be clear from context to which structure an interpretation is part of). Sometimes we also speak of the *extension* of a constant symbol, to mean its value given by an interpretation. Note that constant symbols are never of arity zero, hence the extension of a constant symbol is always a set. Further, as a convention, when we describe some structure using Gothic letters, e.g. $\mathfrak{A}, \mathfrak{B}$, then we simply refer to the underlying domain using an uppercase roman typeface, e.g. A, B, respectively.

We introduce two basic concepts involving structures: isomorphisms and substructures. To do so, it is necessary to transport values of one domain to another domain. Let f be a bijection between the domains A and B. We can use f directly as a bijection between the values in $A[0]$ and $B[0]$. We can construct a lifting of the bijection f to higher-order values inductively. Let $f_1 : A[\alpha_1] \to B[\alpha_1]$, $\ldots$, $f_n : A[\alpha_n] \to B[\alpha_n]$ be bijections on values of arities $\alpha_1, \ldots, \alpha_n$. Then there exists a bijection $f' : A[\alpha_1] \times \ldots \times A[\alpha_n] \to B[\alpha_1] \times \ldots \times B[\alpha_n]$ by mapping each component of the Cartesian product using $f_1, \ldots, f_n$, respectively. Consequently, there exists a bijection $f'' : \mathcal{P}(A[\alpha_1] \times \ldots \times A[\alpha_n]) \to \mathcal{P}(B[\alpha_1] \times \ldots \times B[\alpha_n])$ between the values in $A[(\alpha_1, \ldots, \alpha_n)]$ and $B[(\alpha_1, \ldots, \alpha_n)]$.

Given two structures $\mathfrak{A}$ and $\mathfrak{B}$. The structures are *isomorphic*, written $\mathfrak{A} \cong \mathfrak{B}$, if and only if there is a bijection f between the domains A and B such that $C^{\mathcal{J}} = f''(C^{\mathcal{I}})$ for every constant symbol C, where f'' is f lifted to a bijection between the values of A and B of the arity associated to C, $\mathcal{I}$ is the interpretation of $\mathfrak{A}$, and $\mathcal{J}$ is the interpretation of $\mathfrak{B}$.

The cardinality of the domains of isomorphic structures are equal. In other words, structures with a finite domain are isomorphic only to structures with also a finite domain, and similar for structures with countable or uncountable domains.

Given two structures $\mathfrak{A}$ and $\mathfrak{B}$. Structure $\mathfrak{A}$ is a *substructure* of $\mathfrak{B}$, written $\mathfrak{A} \subseteq \mathfrak{B}$, if and only if $A \subseteq B$ and $C^{\mathcal{I}} = C^{\mathcal{J}} \cap A(\alpha)$ for every constant symbol C, where $\mathcal{I}$ is the interpretation of $\mathfrak{A}$, and $\mathcal{J}$ is the interpretation of $\mathfrak{B}$.

Definition A.2.4 (Valuations)**.** Given a structure $\mathfrak{A}$. A *valuation* ρ of $\mathfrak{A}$ assigns every variable symbol of arity α to a value of A of arity α.

For a variable v^{α}, by $\rho(v^{\alpha})$ we mean the value given to v^{α} by the valuation ρ. Note that if the domain of $\mathfrak{A}$ is empty, there are no first-order values and thus there cannot be a valuation, since we have (infinitely many) first-order variables that need to be assigned a value. Hence, in the context of a valuation, we may assume the domain of $\mathfrak{A}$ to be non-empty.

Given a valuation ρ and a variable v^{α}, if a is a value of A of arity α, then $\rho[v := a]$ is the *updated* valuation obtained so to satisfy the following two equations:

$$\begin{aligned} &\rho[v := a](v) = a \\ &\rho[v := a](w) = \rho(w) \text{ if } v \text{ and } w \text{ are different} \end{aligned}$$

where w is a meta-variable standing for a variable symbol. Note that if v and w have a different arity, they are necessarily different.

Definition A.2.5 (Satisfaction relation)**.** Given a structure $\mathfrak{A}$ and a valuation ρ of $\mathfrak{A}$, and a formula ϕ. The satisfaction relation $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$ is defined inductively on the structure of ϕ:

1. $\mathfrak{A}, \rho \models^{\mathbf{CL}} \bot$ never holds,
2. $\mathfrak{A}, \rho \models^{\mathbf{CL}} (x \doteq y)$ holds iff $\rho(x) = \rho(y)$,
3. $\mathfrak{A}, \rho \models^{\mathbf{CL}} C(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})$ holds iff $(\rho(v_1^{\alpha_1}), \ldots, \rho(v_n^{\alpha_n})) \in P^{\mathcal{I}}$,
4. $\mathfrak{A}, \rho \models^{\mathbf{CL}} w^{\alpha}(v_1^{\alpha_1}, \ldots, v_n^{\alpha_n})$ holds iff $(\rho(v_1^{\alpha_1}), \ldots, \rho(v_n^{\alpha_n})) \in \rho(w)$ where $\alpha = (\alpha_1, \ldots, \alpha_n)$,
5. $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi \to \psi$ holds iff $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$ implies $\mathfrak{A}, \rho \models^{\mathbf{CL}} \psi$,
6. $\mathfrak{A}, \rho \models^{\mathbf{CL}} \forall v^{\alpha}\phi$ holds iff $\mathfrak{A}, \rho[v := a] \models^{\mathbf{CL}} \phi$ holds for every $a \in A[\alpha]$.

The superscript **CL** stands for Classical Logic. Instead of writing $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$, we may also speak of '$\mathfrak{A}$ and ρ (classically) satisfy ϕ', or 'ϕ is (classically) satisfied by $\mathfrak{A}$ and ρ', or 'ϕ is (classically) satisfiable' if there is some $\mathfrak{A}$ and ρ. We may leave out the superscript **CL** or the word 'classically', if no confusion can arise about the logic we use. In the remainder of this section we drop **CL** and 'classically'.

Note that the definition of the satisfaction relation above breaks down for structures with an empty domain, since there does not exist a valuation if the domain is empty. We still, however, have that some formulas could be considered satisfied in such empty structures (e.g. $\bot \to \bot$ and $\forall x \bot$) whereas other formulas should not (e.g. $\forall P(P(x))$). This technical inconvenience can be resolved by introducing a pseudo valuation for use in empty structures only, which does not assign first-order variables a value. However, we shall leave out the tedious details, and keep ourselves to non-empty structures.

Also the abbreviations can be given a semantics, which follow easily from the definition above:

- $\mathfrak{A}, \rho \models^{\mathbf{CL}} \top$ always holds,
- $\mathfrak{A}, \rho \models^{\mathbf{CL}} \neg\phi$ holds iff $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$ does not hold,
- $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi \wedge \psi$ holds iff $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$ and $\mathfrak{A}, \rho \models^{\mathbf{CL}} \psi$ holds,
- $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi \vee \psi$ holds iff $\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi$ or $\mathfrak{A}, \rho \models^{\mathbf{CL}} \psi$ holds,
- $\mathfrak{A}, \rho \models^{\mathbf{CL}} \exists v^{\alpha}\phi$ holds iff $\mathfrak{A}, \rho[v := a] \models^{\mathbf{CL}} \phi$ holds for some $a \in A[\alpha]$.

The satisfaction relation depends on only finitely many variables being assigned a value by a valuation. This is formally captured by the following proposition. Given a set of variables X, by $\rho[X] = \rho'[X]$ we mean that the valuations ρ and ρ' *coincide* on X, that is, ρ and ρ' assign the same values to variables in X.

Proposition A.2.6 (Coincidence condition)**.** *Given that $\rho[FV(\phi)] = \rho'[FV(\phi)]$, it follows that $\mathfrak{A}, \rho \models \phi$ if and only if $\mathfrak{A}, \rho' \models \phi$.*

Similarly, the choice of bound variables bears no significance on the meaning of a formula. This is formally captured by the following proposition.

Proposition A.2.7 (Invariance under renaming)**.** *Given a formula ϕ, and a renaming π such that all free variables of ϕ stay the same, i.e. $\pi(v) = v$ for all $v \in FV(\phi)$. It follows that $\mathfrak{A}, \rho \models \phi$ if and only if $\mathfrak{A}, \rho \models \pi(\phi)$.*

The proposition above is significant, as it provides a semantic justification for Barendregt's variable convention (see Convention A.1.8). It is always possible to rename the bound variables of a formula, so that the bound variables and free variables are separated, without changing the meaning of a formula.

Lemma A.2.8 (Substitution lemma)**.** *Given a formula ϕ and variables v^α, w^α, then $\mathfrak{A}, \rho \models \phi[v^\alpha := w^\alpha]$ if and only if $\mathfrak{A}, \rho[v^\alpha := \rho(w^\alpha)] \models \phi$.*

Sometimes, it is more convenient to work with the set of valuations by which a formula is satisfied given a particular structure.

Definition A.2.9 (Denotation)**.** The *denotation* of a formula $\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$ is defined:

$$\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}} = \{\rho \mid \mathfrak{A}, \rho \models^{\mathbf{CL}} \phi\}.$$

Similar as before, we may drop **CL** if clear from context. We write $\phi \equiv_{\mathfrak{A}} \psi$ for $\mathfrak{A}[\![\phi]\!] = \mathfrak{A}[\![\psi]\!]$, and say that ϕ and ψ are equivalent.

We write $\mathfrak{A} \models \phi$ to mean $\mathfrak{A}, \rho \models \phi$ for all valuations ρ, and we say that ϕ is *true* in $\mathfrak{A}$. If ϕ is a sentence that is satisfied by $\mathfrak{A}$ and some valuation, using the coincidence condition, we can obtain that it is also satisfied by the same structure but with any other valuation: the valuation has no influence on whether a sentence is satisfied by the structure. So if ϕ is a sentence, it is true in $\mathfrak{A}$ if and only if it is satisfied by $\mathfrak{A}$, that is, $\mathfrak{A} \models \phi$ if and only if $\mathfrak{A}, \rho \models \phi$ for some valuation ρ.

Given a sentence ϕ, we write $\models \phi$ to mean that $\mathfrak{A} \models \phi$ for all structures $\mathfrak{A}$, and we then say that ϕ is *valid.*

Given a theory, i.e. a set of sentences Γ, we write $\mathfrak{A} \models \Gamma$ to mean that all sentences in Γ are true in $\mathfrak{A}$, that is, $\mathfrak{A} \models \phi$ for all $\phi \in \Gamma$. We may then also speak of 'Γ is satisfied by $\mathfrak{A}$'. A theory Γ is *satisfiable* if there exists a structure $\mathfrak{A}$ such that $\mathfrak{A} \models \Gamma$. A theory Γ is *finitely satisfiable* if every finite subset of Γ is satisfiable.

Theorem A.2.10 (Compactness)**.** *Given a first-order theory Γ. Γ is satisfiable if and only if Γ is finitely satisfiable.*

Proof. Follows from Łoś's theorem and an ultraproduct construction, see [87, Theorem 2.10]. □

We write $\Gamma \models \phi$ to mean $\mathfrak{A} \models \phi$ for all structures $\mathfrak{A}$ such that $\mathfrak{A} \models \Gamma$, and say that ϕ is a *semantic consequence* of Γ. As an additional case of semantic consequence, we consider a context, i.e. a list of formulas Γ, and a formula ϕ. We write $\Gamma \models \phi$ to mean $\mathfrak{A}, \rho \models \phi$ for all structures $\mathfrak{A}$ and valuations ρ such that $\mathfrak{A}, \rho \models \psi$ for every formula $\psi \in \Gamma$. Note that for contexts, we deal with formulas that may contain free variables. As such, there is one valuation that is used in both checking the satisfaction of all formulas of the context, and in satisfaction of the given formula ϕ. If we have only sentences in Γ and ϕ is also a sentence, then both readings of $\Gamma \models \phi$ coincide.

By $\mathit{Th}^{\mathbf{CL}}(\mathfrak{A})$ we mean the set of all sentences ϕ such that $\mathfrak{A} \models^{\mathbf{CL}} \phi$, and we speak of the *higher-order theory of* $\mathfrak{A}$. (Again, we may drop the superscript **CL** if clear from context.) If we restrict $\mathit{Th}(\mathfrak{A})$ to the first-order formulas, denoted $\mathit{Th}_1(\mathfrak{A})$, we speak of the *first-order theory of* $\mathfrak{A}$. If we restrict $\mathit{Th}(\mathfrak{A})$ to the second-order formulas, denoted $\mathit{Th}_2(\mathfrak{A})$, we speak of the *second-order theory of* $\mathfrak{A}$. And so on for higher orders. By the way we classify formulas, higher-order theories always include lower-order theories, i.e. $\mathit{Th}_1(\mathfrak{A}) \subseteq \mathit{Th}_2(\mathfrak{A}) \subseteq \ldots$.

In general, we have for every structure $\mathfrak{A}$ and formula ϕ that either $\mathfrak{A} \models \phi$ or $\mathfrak{A} \models \neg\phi$. Thus, the (first-order, second-order, ..., higher-order) theory of a structure is necessarily complete.

Given two structures $\mathfrak{A}$ and $\mathfrak{B}$. The structures are *elementarily equivalent*, written $\mathfrak{A} \equiv_1^{\mathbf{CL}} \mathfrak{B}$, if and only if for every first-order sentence ϕ we have $\mathfrak{A} \models^{\mathbf{CL}} \phi$ if and only if $\mathfrak{B} \models^{\mathbf{CL}} \phi$. (We may drop the superscript **CL** under the same proviso.) In other words, two elementarily equivalent structures satisfy exactly the same first-order sentences, i.e. $\mathit{Th}_1(\mathfrak{A}) = \mathit{Th}_1(\mathfrak{B})$.

Given a set of sentences Γ, by $\mathrm{Mod}^{\mathbf{CL}}(\Gamma)$ we mean the class of all structures $\mathfrak{A}$ such that $\mathfrak{A} \models^{\mathbf{CL}} \Gamma$. (Same treatment of the superscript **CL**.) Gaining insight in the classification of structures is the main goal of model theory. A first result of model theory is given below.

Proposition A.2.11. *A first-order theory* Γ *is complete if and only if all structures* $\mathfrak{A}, \mathfrak{B} \in \mathrm{Mod}(\Gamma)$ *are elementarily equivalent, i.e.* $\mathfrak{A} \equiv_1 \mathfrak{B}$.

The class of *finite structures* consists of structures $\mathfrak{A} = (A, \mathcal{I})$ where the domain A is finite. A natural question to ask is: is it possible to give a sentence that characterizes finite structures? To be able to answer that question, it is worthwhile to give an informal proof of the following proposition.

Proposition A.2.12. *A set* D *is finite if and only if every injective total function* $f : D \to D$ *is a surjection.*

Proof. Suppose D is finite, and let f be an injective total function. Suppose, towards contradiction, that f is not a surjection. Then there is an unreachable element, i.e. some x for which there is no input y such that the output $f(y) = x$. Since f is a total function, the function f must be defined for every input: there is exactly one outgoing pointer for every element in D. Since D is finite, there are n points. So there are n pointers, but at most $n - 1$ points are reached. Then, according to the pigeonhole principle, there must be one point which can be reached through f from two different inputs. This is in contradiction with the fact that f is injective.

Suppose every injective total function $f : D \to D$ is a surjection. Suppose, towards contradiction, that D is infinite. Then D must be non-empty, and let d be some element of D (it does not matter which one you choose). Now consider the set $D \setminus \{d\}$, which is still infinite. We can place every element of $D \setminus \{d\}$ next to precisely one element of D, and thus there is a bijection between D and $D \setminus \{d\}$. However, this shows that we have a total function that is an injection from D to D, but not a surjection, since d is never reached. Contradiction! □

Let R be a 2-ary variable, and x, y, z distinct individual variables. We have the following abbreviations:

- $\mathit{fun}(R)$ abbreviates $\forall x, y, z.\ R(x, y) \land R(x, z) \to y \doteq z$,
- $\mathit{inj}(R)$ abbreviates $\forall x, y, z.\ R(x, z) \land R(y, z) \to x \doteq y$,
- $\mathit{tot}(R)$ abbreviates $\forall x \exists y R(x, y)$,
- $\mathit{surj}(R)$ abbreviates $\forall y \exists x R(x, y)$.

Our characterizing sentence is now the following.

Proposition A.2.13 (Characterization of finite structures)**.** *$\mathfrak{A}$ is a finite structure if and only if*

$$\mathfrak{A} \models \forall R.\ \mathit{fun}(R) \land \mathit{tot}(R) \land \mathit{inj}(R) \to \mathit{surj}(R).$$

Proposition A.2.14. *Given structures $\mathfrak{A}$ and $\mathfrak{B}$. Then $\mathfrak{A} \cong \mathfrak{B}$ implies $\mathfrak{A} \equiv_1 \mathfrak{B}$. The converse also holds for finite structures.*

Thus, first-order logic is sufficiently powerful for classifying the finite structures (up to isomorphism).

An important class of structures is that of *countable structures*, which are structures with a countable domain. We say that a set X is *countable* if there exists an *enumeration* function $f : \mathbb{N} \to X \cup \{\bot\}$ from the natural numbers to the set $(X \cup \{\bot\})$ in which $\bot$ is a dummy element not in X, such that for every element $x \in X$ there exists a natural number n such that $f(n) = x$. We make use of a dummy element to ensure that finite sets are also considered countable.

In a countable structure, the first-order values are countable although the second and higher-order values are not countable. Although it is the case that for given countable sets their finitary Cartesian product is again countable, this fails for power sets. For a given countable set its power set is not countable (which follows from a diagonalization argument).

Lemma A.2.15. *Given a finite signature and countable structures $\mathfrak{A}$ and $\mathfrak{B}$. Then $\mathfrak{A} \equiv_2 \mathfrak{B}$ implies $\mathfrak{A} \cong \mathfrak{B}$.*

Proof. The proof requires the axiom of constructibility, see also [5]. □

Thus, second-order logic is sufficiently powerful for classifying the countable structures (up to isomorphism).

If we restrict ourselves to first-order signatures, we also have an important class of structures called data structures. Essentially, a *data structure* is a countable structure with a computable interpretation. Formally, this amounts to the following conditions, where X is the domain:

- there exists an enumeration function $f : \mathbb{N} \to X \cup \{\bot\}$ such that for each $x \in X$ there exists a *unique* natural number n such that $f(n) = x$,
- the set $\{n \mid f(n) \neq \bot\}$ is computable (i.e. it is decidable whether a natural number represents an element of the domain of the data structure or not),
- the interpretation is computable (i.e. the extension of every constant symbol is a decidable set).

The unique natural number corresponding to each element of the domain is called its *encoding*. One uses the encoding of an element in showing that the interpretation is computable, since in data structures one can easily go back and forth between the elements of the domain and their encoding as a natural number.

Proposition A.2.16. *It is decidable whether a quantifier-free formula is satisfied in a given data structure and valuation.*

Note that data structures induce a natural order relation on its element, by their enumeration order. With some additional effort, one could also define bounded formulas (in which existential and universal quantification is always bounded) and extend the above decidability property to bounded formulas as well. Further, given a bounded formula ϕ, it is semi-decidable whether the formula $\exists x\phi$ is satisfied in a given data structure and valuation.

A.3 Basic proof theory

We now investigate syntactical systems for deduction, also called *proof systems*. First, we introduce proof systems in abstracto, in the sense that we abstract from the (syntactic) objects which are involved in deductions. Many interesting properties of proof systems can already be stated in the abstract, regardless of the syntactic objects [26]. Then, we investigate a particular proof system for classical logic, by instantiating objects by the formulas of our assertion language. Later in this thesis, we also introduce proof systems for reasoning about separation logic and program correctness, thus further motivating the approach of giving proof systems in the abstract first.

Definition A.3.1 (Proof system). A *proof system* $\mathfrak{D} = (O, /)$ consists of a class of objects O and a deduction relation $/$ on lists of objects and objects, that satisfy the following conditions:

(Rg) $a_1, \ldots, a_n \,/\, a_i$ for any $1 \leq i \leq n$,

$$\textbf{(Tg)}\quad \left.\begin{array}{ll} a_1, \ldots, a_n \,/\, b_1, & \\ \vdots & \vdots, \\ a_1, \ldots, a_n \,/\, b_m & \end{array}\right\} \text{ and } b_1, \ldots, b_m \,/\, c \text{ implies } a_1, \ldots, a_n \,/\, c.$$

Whenever a list of objects $a_1, \ldots, a_n$ and an object b are related by the deduction relation $/$, we say that 'b follows from $a_1, \ldots, a_n$' or '$a_1, \ldots, a_n$ leads to b'. If that is the case, the objects $a_1, \ldots, a_n$ are called the *premises* and b is called the *conclusion*. The witness that the deduction relation between premises and a conclusion holds is called a *deduction*. It should be clear from context to which proof system the deduction relation $/$ belongs. We fix some proof system $\mathfrak{D} = (O, /)$ until Definition A.3.4.

The condition (Rg) is called *generalized reflexivity*, and the condition (Tg) is called *generalized transitivity*. Note that the we require an expressive meta language due to the many ellipsis: the condition (Tg) reads as 'if b_i follows from $a_1, \ldots, a_n$ for all $1 \leq i \leq m$ and c follows from $b_1, \ldots, b_m$, then c also follows from $a_1, \ldots, a_n$'. Both conditions imply the non-generalized facts:

(R) $a \mathbin{/} a$,

(T) $a \mathbin{/} b$ and $b \mathbin{/} c$ implies $a \mathbin{/} c$,

which establishes that the deduction relation is reflexive and transitive.

In some logic texts, deductions are depicted in a different way. Instead of writing $a_1, \ldots, a_n \mathbin{/} b_1$ up to $a_1, \ldots, a_n \mathbin{/} b_m$, deductions are rendered as

$$\begin{array}{ccc} a_1 \ldots a_n & & a_1 \ldots a_n \\ \mathcal{D}_1 & & \mathcal{D}_m \\ b_1 & \ldots & b_m \end{array}$$

Deductions are tree shaped, where a conclusion is at the root of the tree and the premises are its leaves. Note that in the depiction above the premises and the conclusions are all part of the deductions. So, above, $a_1, \ldots, a_n$ and b_1 are all part of $\mathcal{D}_1$. With this perspective in mind, we may also call deductions *proof trees*. Using this notation we can depict generalized transitivity as follows. Assuming the deduction given above, and the deduction given below

$$\begin{array}{c} b_1 \ldots b_m \\ \mathcal{D}' \\ c \end{array}$$

we can imagine pasting the proof trees $\mathcal{D}_1$ up to $\mathcal{D}_n$ in the place of the leaves $b_1, \ldots, b_m$ of the proof tree of $\mathcal{D}'$, where the conclusions of the former proof trees overlap with the premises of the latter, to finally obtain the deduction:

$$\begin{array}{ccc} a_1 \ldots a_n & & a_1 \ldots a_n \\ \mathcal{D}_1 & & \mathcal{D}_m \\ b_1 & \ldots & b_m \\ & \mathcal{D}' & \\ & c & \end{array}$$

which is to say, there exists a deduction $\mathcal{D}$:

$$\begin{array}{c} a_1 \ldots a_n \\ \mathcal{D} \\ c \end{array}$$

A proof system is called *finitary* if the class of objects is a decidable set and if there are finitary means to establish that the deduction relation holds. We do not require that the class of objects is a finite set, but we do require a recursive deduction relation. This can be imagined by having finite *certificates* that serve as witnesses for establishing that the deduction relation holds. Finitary proof systems play an essential rôle in computer science, since the certificates that establish deductions of a finitary proof system can be checked by a computer. This allows for the development of tools for constructing and checking deductions. In the remainder, we shall pay attention mostly to finitary proof systems.

Lemma A.3.2 (Exchange, weakening, contraction)**.**

(E) $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n \,/\, b$ *implies* $a_1, \ldots, a_{i+1}, a_i, \ldots, a_n \,/\, b$*,*

(W) $a_1, \ldots, a_n \,/\, c$ *implies* $a_1, \ldots, a_n, b_1, \ldots, b_m \,/\, c$*,*

(C) $a_1, \ldots, a_n, b, b, c_1, \ldots, c_m \,/\, d$ *implies* $a_1, \ldots, a_n, b, c_1, \ldots, c_m \,/\, d$*.*

The proof follows easily from generalized reflexivity and transitivity. The conditional (E) is called the property of *exchange.* It describes, intuitively, that if there is a deduction from a list of premises, then we must also have a deduction in which the premises are permuted. The conditional (W) is called the property of *weakening.* Intuitively it says, if there is a deduction from a list of premises, we must also have a deduction in which additional (but unused) premises are present. Finally, the conditional (C) is called the property of *contraction.* Intuitively, the multiplicity of premises do not matter. Thus, it is possible to see the list of premises of any deduction as a finite set of objects (where duplicates and order do not matter), which can always be extended with additional premises.

We now introduce concepts that are derived from the deduction relation. If we have two objects and one object follows from the other and vice versa, then we say that the two objects are mutually deducible. For that purpose we introduce the following abbreviation,

$$a \,/\!/\, b \text{ abbreviates } a \,/\, b \text{ and } b \,/\, a$$

Mutual deducibility has two important properties, namely that we can replace any conclusion or premise with an object which is mutually deducible. The proof of the following lemma is again simple.

Lemma A.3.3 (Substitutivity)**.** $a \,/\!/\, b$ *and* $c_1, \ldots, c_n \,/\, a$ *implies* $c_1, \ldots, c_n \,/\, b$*,* $a \,/\!/\, b$ *and* $c_1, \ldots, c_n, a, d_1, \ldots, d_m \,/\, e$ *implies* $c_1, \ldots, c_n, b, d_1, \ldots, d_m \,/\, e$*.*

An important derived concept is that of *relative demonstrability.* When describing proof systems, one is foremost interested in this concept. We introduce the following notation: $\vdash^{\mathfrak{D}}$. The symbol $\vdash$ is called a *turnstile.* The superscript annotates which proof system is used, and may be dropped if the proof system is clear from context. We first define the concept of relative demonstrability, which can then be refined into four concepts familiar to most users of logic: complementarity, demonstrability, contradictoriness, and refutability.

Definition A.3.4 (Relative demonstrability)**.** Let $\mathfrak{D} = (O, /)$ be a proof system. We define *relative demonstrability* as a relation $\vdash^{\mathfrak{D}}$ on lists of objects, as follows: $a_1, \ldots, a_n \vdash^{\mathfrak{D}} b_1, \ldots, b_m$ if and only if

$$b_1, d_1, \ldots, d_k \mathbin{/} c \text{ and } \ldots \text{ and } b_m, d_1, \ldots, d_k \mathbin{/} c \text{ implies } a_1, \ldots, a_n, d_1, \ldots, d_k \mathbin{/} c$$

for all $d_1, \ldots, d_k$ and for all c.

Given that a list of objects $a_1, \ldots, a_n$ and a list of objects $b_1, \ldots, b_m$ are related by the relative demonstrability relation, i.e. $a_1, \ldots, a_n \vdash b_1, \ldots, b_m$, then we call the objects $a_1, \ldots, a_n$ the *antecedents* and the objects $b_1, \ldots, b_m$ the *succedents.* We shall talk about the exceptional cases, when either of the two lists is empty, as follows: if $\vdash b_1, \ldots, b_m$ then we call $b_1, \ldots, b_m$ *complementary*, if $\vdash b$ then we call b *provable*, if $a_1, \ldots, a_n \vdash$ then we call $a_1, \ldots, a_n$ *contradictory*, and if $a \vdash$ then we call a *refutable.*

We have the following important property of relative demonstrability:

$$a_1, \ldots, a_n \vdash b \text{ if and only if } a_1, \ldots, a_n \mathbin{/} b.$$

It captures that relative demonstrability and the deduction relation coincide in case there is a single succedent. Furthermore, we have

$$a_1, \ldots, a_n \mathbin{/} b_1 \text{ or } \ldots \text{ or } a_1, \ldots, a_n \mathbin{/} b_m \text{ implies } a_1, \ldots, a_n \vdash b_1, \ldots, b_m.$$

Note that the converse does not hold in general. While the deduction relation must be recursive, relative demonstrability is not necessarily recursive (this can be seen from its definition, where we have an unbounded universal quantification over sequences of objects).

Example A.3.5. Construct a proof system: take O to be two distinct objects, say a and b, and take the smallest deduction relation that satisfies generalized reflexivity (and thus generalized transitivity). Then we do have that $/a$ or $/b$ implies $\vdash a, b$. But the converse fails. Clearly $\vdash a, b$ holds (consider on the meta-level that only instances of generalized reflexivity satisfy the premise when both a and b are in $d_1, \ldots, d_n$). But we have neither $/a$ nor $/b$.

Similar to Lemma A.3.2 we also have properties of exchange, weakening, and contraction, but on either sides of the turnstile.

Lemma A.3.6 (Left/right exchange, weakening, contraction)**.**

(LE) $a_1, \ldots, a_i, a_{i+1}, \ldots, a_n \vdash b_1, \ldots, b_m$ *implies* $a_1, \ldots, a_{i+1}, a_i, \ldots, a_n \vdash b_1, \ldots, b_m$,

(RE) $a_1, \ldots, a_n \vdash b_1, \ldots, b_i, b_{i+1}, \ldots, b_m$ *implies* $a_1, \ldots, a_n \vdash b_1, \ldots, b_{i+1}, b_i, \ldots, b_m$,

(LW) $a_1, \ldots, a_n \vdash c_1, \ldots, c_k$ *implies* $a_1, \ldots, a_n, b_1, \ldots, b_m \vdash c_1, \ldots, c_k$,

(RW) $a_1, \ldots, a_n \vdash c_1, \ldots, c_k$ *implies* $a_1, \ldots, a_n \vdash c_1, \ldots, c_k, b_1, \ldots, b_m$,

(LC) $a_1, \dots, a_n, b, b, c_1 \dots, c_m \vdash d_1, \dots, d_k$ *implies*
$a_1, \dots, a_n, b, c_1 \dots, c_m \vdash d_1, \dots, d_k$,

(RC) $a_1, \dots, a_n \vdash c_1, \dots, c_m, b, b, d_1, \dots, d_k$ *implies*
$a_1, \dots, a_n \vdash c_1, \dots, c_m, b, d_1, \dots, d_k$.

Note that an easy corollary that follows from right exchange and right weakening is that we also have generalized reflexivity for $\vdash$, as follows:

$$a_1, \dots, a_n \vdash b_1, \dots, b_m \text{ if } a_i = b_j \text{ for some } i, j.$$

Another important consequence of the definition of relative demonstrability is:

Lemma A.3.7 (Cut)**.** *If* $a_1, \dots, a_n \vdash b_1, \dots, b_m, e$ *and* $e, a_1, \dots, a_n \vdash b_1, \dots, b_m$ *then* $a_1, \dots, a_n \vdash b_1, \dots, b_m$.

Proof. Fix arbitrary $d_1, \dots, d_k$ and c. It is sufficient, assuming $b_i, d_1, \dots, d_k \,/\, c$ for $1 \leq i \leq m$, to show $a_1, \dots, a_n, d_1, \dots, d_k \,/\, c$. Applying (LW) and (LE) on our assumptions, we obtain $b_i, a_1, \dots, a_n, d_1, \dots, d_k \,/\, c$ for $1 \leq i \leq m$. From the second premise we know that $e, a_1, \dots, a_n, d_1, \dots, d_k \,/\, c$. Applying all these facts to the first premise we obtain $a_1, \dots, a_n, a_1, \dots, a_n, d_1, \dots, d_k \,/\, c$. After applying (LE) and (LC) we reach our goal. □

Now that we have explored proof systems in the abstract, we can construct particular and concrete proof systems. Typically, one constructs a finitary proof system by following three steps:

1. One first specifies what is the class of objects. The class of objects typically has a certain structure, e.g. there are operations defined on objects such that the class of objects is closed under application of the operations.

2. One defines the deduction relation: this can be done by introducing *axioms* and *proof rules*, often in the form of axiom and proof rule *schemata*.

3. One checks that the resulting proof system is finitary (i.e. the class of objects and the deduction relation are recursive), by showing there is an algorithm that can decide the deduction relation.

The third step is easy if one takes a recursive set of objects and defines the deduction relation inductively. However, an alternative to the second step above is by imposing constraints on the deduction relation, e.g. in terms of relative demonstrability. Then the third step is non-trivial.

We consider two classes of proof rules: simple and complex. Axioms and simple proof rules are often depicted as follows:

$$\frac{}{b} \qquad \frac{a_1 \quad \dots \quad a_n}{b}$$

where the axiom on the left denotes $\vdash b$ (the conclusion b follows from no premises) and the proof rule on the right denotes $a_1, \dots, a_n \vdash b$ (the conclusion follows from the premises $a_1, \ldots, a_n$). One may consider an axiom to be a proof rule without premises. Contrastingly, one may depict complex proof rules as follows:

$$\dfrac{\begin{matrix}\mathcal{D}_1 & & \mathcal{D}_n \\ a_1 & \ldots & a_n\end{matrix}}{b}$$

which expresses how to construct a deduction with conclusion b, given deductions $\mathcal{D}_1$ up to $\mathcal{D}_n$ with conclusions a_1 up to a_n, respectively. In such constructions, one also has to describe how to treat the premises of the deductions on top of the rule. For example, the constructed deduction may have less premises than the premises of $\mathcal{D}_1$ up to $\mathcal{D}_n$ combined. In such cases, the object used as premise in the deduction on top is called an *assumption*, which is *closed* by the complex proof rule. Such situations are also depicted as follows:

$$\dfrac{a_1 \quad \ldots \quad \boxed{\begin{matrix} c_1 \\ \ldots \\ c_k \\ \\ a_n \end{matrix}}}{b}$$

to indicate that the premises of the deduction on top of a_1 are also taken to be premises of the resulting deduction, but that the objects $c_1, \ldots, c_k$ in the deduction on top of a_n are assumptions, and hence not premises of the resulting deduction.

A simple proof rule can be considered to be a complex proof rule where the premises are shared among all deductions and no assumptions are closed: this is what generalized transitivity ensures. Axiom and (simple or complex) proof rule schemata employ meta-linguistic constructs to describe constraints that must hold for its instances.

Example A.3.8. Consider the set $\mathbb{N}$ of objects, where 0 is zero and $s : \mathbb{N} \to \mathbb{N}$ the successor function. We have the following axiom and simple proof rule schema:

$$\dfrac{}{0} \qquad \dfrac{x}{s(x)}$$

The proof rule is a schema, where x is a meta-linguistic variable standing for any object $x \in \mathbb{N}$, and $s(x)$ is the object obtained after applying the successor function s at the meta-level. The deduction relation can now inductively be defined: only the above axiom and proof rules (instances of the above proof rule schema) may be employed, next to generalized reflexivity and generalized transitivity that hold for every proof system. As an example, we have $1 \,/\, 3$ as shown by its deduction:

$$\dfrac{\dfrac{1}{2}}{3}$$

which we read as stating that both 2 follows from 1 (that is, $1 \,/\, 2$), and 3 follows from 2 (that is, $2 \,/\, 3$), which can be combined in one deduction by transitivity. The above is one of the many possible deductions establishing the same conclusion (in this case 3) with the same premises (in this case just 1). *End of Example.*

An important aspect of proof theory is the analysis of proof systems. Recall, if we have $\vdash a$, we say that a is *provable*. Given a proof system $\mathfrak{D}$, one could consider the class of all provable objects $\{a \mid \vdash^{\mathfrak{D}} a\}$. Proof systems can be compared by comparing their classes of provable objects. In particular, one can ask whether a given proof rule is redundant. A proof rule is redundant if in a proof system *without* that proof rule, the class of provable objects is the same as for the proof system *with* that proof rule. We introduce two concepts that capture such redundancy, in essentially different ways: *derivability* and *admissibility* of proof rules.

Given a proof system $\mathfrak{D}$ which has a simple proof rule

$$\frac{a_1 \quad \ldots \quad a_n}{b}$$

then if $a_1, \ldots, a_n \vdash^{\mathfrak{D}^-} b$ in a proof system $\mathfrak{D}^-$ without the above proof rule, we call that proof rule *derivable*. Clearly, a derivable proof rule is redundant, since the class of all provable objects of $\mathfrak{D}$ and $\mathfrak{D}^-$ are the same. Indeed, for any deduction in $\mathfrak{D}$ where the rule is used, we can 'cut' out the rule and 'paste' in the deduction witnessing $a_1, \ldots, a_n \vdash^{\mathfrak{D}^-} b$ at the place where the rule is used. If this 'cut and paste'-procedure is applied for every occurrence of the proof rule, one ends up with a deduction in $\mathfrak{D}^-$.

Similarly, given a proof system $\mathfrak{D}$ which has a complex proof rule

$$\frac{\begin{matrix} \mathcal{D}_1 & & \mathcal{D}_n \\ a_1 & \ldots & a_n \end{matrix}}{b}$$

then if $c_1, \ldots, c_{m_i} \vdash^{\mathfrak{D}^-} a_i$ for all $1 \leq i \leq n$ implies $d_1, \ldots, d_k \vdash^{\mathfrak{D}^-} b$ in a proof system $\mathfrak{D}^-$ without the above proof rule, we call that proof rule *admissible* (the complex proof rule imposes conditions on how the premises of the involved deductions $c_1, \ldots, c_{m_i}$ are related to the premises of the constructed deduction $d_1, \ldots, d_k$). Informally, a complex proof rule is admissible if it can be mimicked by a construction involving the deductions $\mathcal{D}_1$ up to $\mathcal{D}_n$. Also an admissible proof rule is redundant. Consider a deduction of $\mathfrak{D}$ in which the above complex proof rule is used. Consider the context where the proof rule occurs, such that the proof rule does not occur in the deductions of the premises, but each deduction $\mathcal{D}_i$ has a list of premises $c_1, \ldots, c_{m_i}$. We cut out the derivations with as conclusion the premises $a_1, \ldots, a_n$, to establish (possibly after weakening) $c_1, \ldots, c_{m_1} \vdash^{\mathfrak{D}^-} a_1$ and ... and $c_1, \ldots, c_{m_n} \vdash^{\mathfrak{D}^-} a_n$, from which we then obtain a new deduction by the fact that the rule is admissible, which can be placed in the place of the rule. After this procedure is applied for every occurrence of the proof rule, from the leaves to the root, one ends up with a deduction in $\mathfrak{D}^-$.

A proof rule is *weakly admissible* if we have that $\vdash^{\mathfrak{D}^-} a_i$ for all $1 \leq i \leq n$ implies $\vdash^{\mathfrak{D}^-} b$. Thus weakly admissible proof rules are not necessarily eliminable in arbitrary contexts. If a proof rule is not weakly admissible, it is not admissible either. Further, in some cases, the requirement of admissible proof rules that the resulting deduction $d_1, \ldots, d_k \vdash^{\mathfrak{D}^-} b$ does not contain the proof rule *at all* is too strong. If our goal is to eliminate the rule from any deduction, it is sufficient that the proof rule can be pushed upward in every deduction in which it occurs, and eliminated when it occurs near the top: so that either the size of the deductions used as premises become smaller, or the resulting deduction is indeed in $\mathfrak{D}^{-}$.

When comparing derivability with admissibility, there is an important distinction: derivability can be applied to any rule occurrence, but admissibility can only be applied to rule occurrences where the deductions of the premises are already free from occurrences of the redundant rule. In admissibility, the resulting deduction with the redundant rule eliminated is constructed from the leaves back to the root, whereas no such order is imposed when eliminating derivable rules. Moreover, to obtain a deduction where the redundant rule is eliminated, for admissible proof rules, the original deductions of the premises can be changed (except for for their premises and conclusion), whereas for derivable rules those deductions remain intact.

In fact, derivability of a proof rule is stronger than admissibility. This can be demonstrated by the following example, where we analyze a proof rule with conclusion b and a single premise a. Consider the following two properties:

$$\text{if } a \,/\, b \text{ then } \vdash a \text{ implies } \vdash b \tag{A.1}$$

$$\text{if } \vdash a \text{ implies } \vdash b \text{ then } a \,/\, b \tag{A.2}$$

The first property (A.1) states that derivability implies weak admissibility. Given a deduction $a \,/\, b$, we know that the provability of a implies the provability of b. This is a consequence of transitivity, by applying the deduction without premises of a in the place of the premise a in the deduction $a \,/\, b$ to obtain $\vdash b$. This argument also works when there are arbitrary additional premises $c_1, \dots, c_m$ from which a follows, hence derivability also implies admissibility. Hence, the first property holds for every proof system.

The second property (A.2) states that weak admissibility implies derivability. However it can be shown that the second property does not hold in general: there is a proof system in which it fails. Hence, admissibility does not imply derivability, since weak admissibility already does not imply derivability.

Example A.3.9. Consider the set $\mathbb{Z}$ of objects, where 0 is zero, $s : \mathbb{Z} \to \mathbb{Z}$ the successor function, and $p : \mathbb{Z} \to \mathbb{Z}$ the predecessor function. Take the same axiom and proof rule schema as we did in the last example (so we have only the instances where $x \in \mathbb{N}$):

$$\frac{}{0} \qquad \frac{x}{s(x)}$$

Comparing this proof system to the previous example, we see that their sets of provable objects must be the same. We now consider the simple proof rule schema

$$\frac{p(x)}{x}$$

with the question: is this proof rule admissible? Consider a deduction and the top-most occurrence of an instance of this proof rule (i.e. those occurrences where deductions of the premise do not have an instance of this proof rule as an occurrence).

Then it must be the case that the conclusion $p(x)$ is in $\mathbb{N}$. We can then eliminate the proof rule by replacing its instance by an instance of the other rule, which deduces $s(p(x))$ from $p(x)$. Keep working downwards and we eventually obtain a deduction in which this proof rule no longer occurs. Hence the proof rule is admissible.

But is it derivable? Consider the following instance: 0 follows from -1. Since the instances of the remaining proof rule are limited to $x \in \mathbb{N}$, we cannot apply it to construct a deduction with the conclusion 0. Hence the new proof rule is not derivable. *End of Example.*

When considering the relative demonstrability relation $\vdash^{\mathfrak{D}}$ of a proof system $\mathfrak{D} = (O, /)$, one typically considers Γ to denote a sequence of objects in the case of $\Gamma \vdash^{\mathfrak{D}} a$. It is natural to extend the relative demonstrability relation to sets of objects too.

Definition A.3.10. Let $\Gamma \subseteq O$ be a set of objects. Then $\Gamma \vdash^{\mathfrak{D}} a$ holds if and only if there exists a sequence Γ_0 of elements in Γ such that $\Gamma_0 \vdash^{\mathfrak{D}} a$.

In fact, we can generalize the succedent too.

Definition A.3.11. Let $\Gamma, \Delta \subseteq O$ be sets of objects. Then $\Gamma \vdash^{\mathfrak{D}} \Delta$ holds if and only if there exists sequence Γ_0 of elements in Γ and sequence Δ_0 of elements in Δ such that $\Gamma_0 \vdash^{\mathfrak{D}} \Delta_0$.

We construct a proof system for first-order classical logic in the style of Hilbert. Let Γ be a context (a finite sequence) of first-order formulas, and ϕ, ψ, ξ be first-order formulas.

Definition A.3.12. Let **CL** be a proof system consisting of:

1. the first-order formulas of classical logic as objects,
2. the smallest deduction relation $\vdash^{\mathbf{CL}}$ satisfying the conditions:

   (MP) $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$ and $\Gamma \vdash^{\mathbf{CL}} \phi$ implies $\Gamma \vdash^{\mathbf{CL}} \psi$,

   (G) $\Gamma \vdash^{\mathbf{CL}} \phi$ implies $\Gamma \vdash^{\mathbf{CL}} (\forall y \, (^{x}_{y}) \, \phi)$
   where $x \notin FV(\Gamma)$ and either $y = x$ or $y \notin FV(\phi)$,

   (A1) $\vdash^{\mathbf{CL}} (\phi \to (\psi \to \phi))$,

   (A2) $\vdash^{\mathbf{CL}} ((\phi \to \psi) \to ((\phi \to (\psi \to \xi)) \to (\phi \to \xi)))$,

   (A3) $\vdash^{\mathbf{CL}} ((\forall x(\phi \to \psi)) \to (\phi \to (\forall y \, (^{x}_{y}) \, \psi)))$
   where $x \notin FV(\phi)$ and either $y = x$ or $y \notin FV(\psi)$,

   (DN) $\vdash^{\mathbf{CL}} (\neg\neg\phi \to \phi)$,

   (∀E) $\vdash^{\mathbf{CL}} ((\forall x \phi) \to (^{x}_{y}) \, \phi)$ where y remains free for x in ϕ,

   (=I) $\vdash^{\mathbf{CL}} (x \doteq x)$,

   (=E) $\vdash^{\mathbf{CL}} ((x \doteq y) \to ((^{z}_{x}) \, \phi \to (^{z}_{y}) \, \phi))$ where x, y remain free for z in ϕ.

It is easy to see that the above proof system is finitary. Each first-order formula is a finite object. Further, the witness of a deduction, $\Gamma \vdash^{\mathbf{CL}} \phi$, is a proof tree with premises in Γ and conclusion ϕ. The leaves of the proof tree are instances of an axiom scheme or a premise in Γ, and the internal nodes of the proof tree are either obtained from the proof rule (MP) where there are two branches, or the proof rule (G) where there is one branch. These proof trees also satisfy generalized reflexivity and generalized transitivity: compositions of proof trees are themselves proof trees.

This proof system is in the style of Hilbert, since the only proof rules are *modus ponens* (MP) and *generalization* (G). Proof rule (MP) is also called implication elimination (→E), and (G) is also called universal introduction (∀I). Note the distinction between (MP) stated above with additional premises in the context Γ, and the stronger condition below:

$$\vdash^{\mathbf{CL}} (\phi \to \psi) \text{ and } \vdash^{\mathbf{CL}} \phi \text{ implies } \vdash^{\mathbf{CL}} \psi \qquad \text{(MP')}$$

which is stated only at the level of provability, i.e. without context. We call (MP') a 'rule of provability', whereas (MP) is called a 'proof rule' (since it is equivalent to $(\phi \to \psi), \phi \vdash^{\mathbf{CL}} \psi$). The difference between these two is that only for a proof system with (MP) we can establish the following property, since in a proof system that has (MP') instead of (MP) we have only deductions from premises obtained from generalized reflexivity or generalization.

Lemma A.3.13. $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$ *implies* $\Gamma, \phi \vdash^{\mathbf{CL}} \psi$.

Proof. Applying weakening we obtain $\Gamma, \phi \vdash^{\mathbf{CL}} (\phi \to \psi)$, and by generalized reflexivity we have $\Gamma, \phi \vdash^{\mathbf{CL}} \phi$. Hence by (MP) we obtain $\Gamma, \phi \vdash^{\mathbf{CL}} \psi$. □

In fact, the converse also holds.

Theorem A.3.14 (Deduction theorem)**.** *If* $\Gamma, \phi \vdash^{\mathbf{CL}} \psi$ *then* $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$.

Proof. Consider the proof tree corresponding to $\Gamma, \phi \vdash^{\mathbf{CL}} \psi$, and perform induction on the structure of that tree. Either the proof tree is a leave (base case), or it is an instance of the (MP) proof rule with two smaller proof trees on top, or an instance of the (G) proof rule with one smaller proof tree on top.

Base case 1. If $\phi = \psi$ then we obtain $\Gamma \vdash^{\mathbf{CL}} (\phi \to \phi)$ from (MP), (A1) and (A2).

Base case 2. If ψ was obtained by reflexivity from Γ or ψ is an instance of an axiom scheme, then we obtain $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$ from (MP) and (A1).

Induction step (MP). Let ψ be the conclusion, where $\Gamma, \phi \vdash^{\mathbf{CL}} (\phi' \to \psi)$ and $\Gamma, \phi \vdash^{\mathbf{CL}} \phi'$ are on top. Our induction hypotheses are $\Gamma, \phi \vdash^{\mathbf{CL}} (\phi' \to \psi)$ implies $\Gamma \vdash^{\mathbf{CL}} (\phi \to (\phi' \to \psi))$, and $\Gamma, \phi \vdash^{\mathbf{CL}} \phi'$ implies $\Gamma \vdash^{\mathbf{CL}} (\phi \to \phi')$. Then we obtain $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$ from (MP) and (A2).

Induction step (G). Let $\psi = (\forall y \binom{x}{y} \psi')$ be the conclusion, where $\Gamma, \phi \vdash^{\mathbf{CL}} \psi'$ is on top and $x \notin FV(\phi)$. Our induction hypothesis is $\Gamma, \phi \vdash^{\mathbf{CL}} \psi'$ implies $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi')$. We obtain the result from (MP) and (A3). □

The deduction theorem proves that the following proof rule is admissible:

$$\frac{\boxed{\begin{array}{c}\phi\\ \\ \psi\end{array}}}{\phi \to \psi}\ (\to\mathrm{I})$$

where the deduction (inside the box) with conclusion ψ may use the assumption ϕ, which may not be a premise of the overall deduction. The ϕ at the top of the box means that the assumption ϕ is closed, that is, although ϕ is a premise of the inner deduction it is not a premise in the resulting, outer deduction. Another consequence of the deduction theorem is that we can clear out the context. Let $\Gamma = \phi_1, \ldots, \phi_n$, then $(\Gamma \to \psi)$ is abbreviates $(\phi_1 \to (\ldots \to (\phi_n \to \psi) \ldots))$. In case Γ is an empty sequence, then $(\Gamma \to \psi)$ is just ψ. Note that this does not work for arbitrary sets, since our formulas are finitary.

Corollary A.3.15. $\Gamma \vdash^{\mathbf{CL}} \psi$ *if and only if* $\vdash^{\mathbf{CL}} (\Gamma \to \psi)$.

Lemma A.3.16. *We have the following derived axioms and proof rules:*

($\top$I) $\vdash^{\mathbf{CL}} \top$,

($\bot$E) $\bot \vdash^{\mathbf{CL}} \phi$,

(DN) $\neg\neg\phi \vdash^{\mathbf{CL}} \phi$,

($\to$I) $\Gamma, \phi \vdash^{\mathbf{CL}} \psi$ *implies* $\Gamma \vdash^{\mathbf{CL}} (\phi \to \psi)$,

($\to$E) $(\phi \to \psi), \phi \vdash^{\mathbf{CL}} \psi$,

($\wedge$I) $\phi, \psi \vdash^{\mathbf{CL}} (\phi \wedge \psi)$,

($\wedge$EL) $(\phi \wedge \psi) \vdash^{\mathbf{CL}} \phi$,

($\wedge$ER) $(\phi \wedge \psi) \vdash^{\mathbf{CL}} \psi$,

($\vee$IL) $\phi \vdash^{\mathbf{CL}} (\phi \vee \psi)$,

($\vee$IR) $\psi \vdash^{\mathbf{CL}} (\phi \vee \psi)$,

($\vee$E) $(\phi \to \xi), (\psi \to \xi), (\phi \vee \psi) \vdash^{\mathbf{CL}} \xi$,

($\forall$I) $\Gamma \vdash^{\mathbf{CL}} \phi$ *implies* $\Gamma \vdash^{\mathbf{CL}} (\forall y \binom{x}{y} \phi)$
where $x \notin FV(\Gamma)$ *and either* $y = x$ *or* $y \notin FV(\phi)$,

($\forall$E) $(\forall x \phi) \vdash^{\mathbf{CL}} \binom{x}{y} \phi$ *where* y *remains free for* x *in* ϕ,

($\exists$I) $\binom{x}{y} \phi \vdash^{\mathbf{CL}} (\exists x \phi)$ *where* y *remains free for* x *in* ϕ,

($\exists$E) $\Gamma \vdash^{\mathbf{CL}} (\exists x \phi)$ *and* $\Gamma, \binom{x}{y} \phi \vdash^{\mathbf{CL}} \psi$ *implies* $\Gamma \vdash^{\mathbf{CL}} \psi$
where $y \notin FV(\Gamma, \psi)$ *and either* $x = y$ *or* $x \notin FV(\phi)$,

(=I) $\vdash^{\mathbf{CL}} (x \doteq x)$,

(=E) $(x \doteq y) \vdash^{\mathbf{CL}} ((\tbinom{z}{x})\, \phi \to (\tbinom{z}{y})\, \phi)$ *where* x, y *remain free for* z *in* ϕ.

Proof. See *Basic Proof Theory* by Troelstra and Schwitchenberg, Section 2.4. □

The proof system consisting of the axiom and proof rules of Lemma A.3.16 is called *natural deduction*. From this proof system it is also possible to derive the axioms and proof rules of Definition A.3.12.

Further, employing the fact that relative demonstrability is a relation on two lists of formulas, we have the following properties. In the following, let Γ and Δ be contexts (finite sequences of formulas). The antecedent context can be seen as a conjunction of its formulas, and the succedent context can be seen as a disjunction of its formulas.

Corollary A.3.17. $\phi, \psi \vdash^{\mathbf{CL}} \xi$ *if and only if* $(\phi \wedge \psi) \vdash^{\mathbf{CL}} \xi$.

Let $\Gamma = \phi_1, \ldots, \phi_n$, then $\bigwedge \Gamma$ is an abbreviation for the formula $(\phi_1 \wedge (\ldots \wedge (\phi_n \wedge \top) \ldots))$. In case Γ is empty, $\bigwedge \Gamma$ is just $\top$.

Corollary A.3.18. $\Gamma \vdash^{\mathbf{CL}} \Delta$ *if and only if* $\bigwedge \Gamma \vdash^{\mathbf{CL}} \Delta$.

Lemma A.3.19. $\Gamma \vdash^{\mathbf{CL}} \phi, \psi$ *if and only if* $\Gamma \vdash^{\mathbf{CL}} (\phi \vee \psi)$.

Proof. Given $\Gamma \vdash^{\mathbf{CL}} \phi, \psi$, and apply ($\vee$IL) and ($\vee$IR), to obtain $\Gamma \vdash^{\mathbf{CL}} (\phi \vee \psi)$. The other direction is more interesting. Given $\Gamma \vdash^{\mathbf{CL}} (\phi \vee \psi)$, and let Δ and ξ be arbitrary. We assume $\phi, \Delta \vdash^{\mathbf{CL}} \xi$ and $\psi, \Delta \vdash^{\mathbf{CL}} \xi$. From these, we have $\phi \vdash^{\mathbf{CL}} (\Delta \to \xi)$ and $\psi \vdash^{\mathbf{CL}} (\Delta \to \xi)$ by Corollary A.3.15. Hence $\Gamma \vdash^{\mathbf{CL}} (\Delta \to \xi)$ by ($\vee$E), and thus $\Gamma, \Delta \vdash^{\mathbf{CL}} \xi$. □

Let $\Gamma = \phi_1, \ldots, \phi_n$, then $\bigvee \Gamma$ is an abbreviation for the formula $(\phi_1 \vee (\ldots \vee (\phi_n \vee \bot) \ldots))$. In case Γ is empty, $\bigvee \Gamma$ is just $\bot$.

Corollary A.3.20. $\Gamma \vdash^{\mathbf{CL}} \Delta$ *if and only if* $\Gamma \vdash^{\mathbf{CL}} \bigvee \Delta$.

The observations above together motivate the introduction of another abbreviation. Let $\Gamma \Rightarrow \Delta$ abbreviate $\bigwedge \Gamma \to \bigvee \Delta$. We call $\Gamma \Rightarrow \Delta$ a *sequent*.

Lemma A.3.21. $\Gamma \vdash^{\mathbf{CL}} \Delta$ *if and only if* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta$.

Lemma A.3.22. *We have the following derived axioms and rules of proof:*

(L$\bot$) $\vdash^{\mathbf{CL}} \bot, \Gamma \Rightarrow \Delta$,

(L$\wedge$) $\vdash^{\mathbf{CL}} \phi, \psi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} (\phi \wedge \psi), \Gamma \Rightarrow \Delta$,

(R$\wedge$) $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \phi$ *and* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \psi$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, (\phi \wedge \psi)$,

(L$\vee$) $\vdash^{\mathbf{CL}} \phi, \Gamma \Rightarrow \Delta$ *and* $\vdash^{\mathbf{CL}} \psi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} (\phi \vee \psi), \Gamma \Rightarrow \Delta$,

(R$\vee$) $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \phi, \psi$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, (\phi \vee \psi)$,

(L→) $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \phi$ *and* $\vdash^{\mathbf{CL}} \psi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} (\phi \to \psi), \Gamma \Rightarrow \Delta$,

(R→) $\vdash^{\mathbf{CL}} \phi, \Gamma \Rightarrow \Delta, \psi$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, (\phi \to \psi)$,

(L∀) $\vdash^{\mathbf{CL}} (\forall x \phi), \binom{x}{y} \phi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} (\forall x \phi), \Gamma \Rightarrow \Delta$
where y remains free for x in ϕ,

(R∀) $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \binom{x}{y} \phi$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, (\forall x \phi)$ *where y is fresh,*

(L∃) $\vdash^{\mathbf{CL}} \binom{x}{y} \phi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} (\exists x \phi), \Gamma \Rightarrow \Delta$ *where y is fresh,*

(R∃) $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, \binom{x}{y} \phi, (\exists x \phi)$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta, (\exists x \phi)$
where y remains free for x in ϕ,

(Ref) $\vdash^{\mathbf{CL}} x \doteq x, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} \Gamma \Rightarrow \Delta$,

(Rep) $\vdash^{\mathbf{CL}} x \doteq y, \binom{z}{y} \phi, \binom{z}{x} \phi, \Gamma \Rightarrow \Delta$ *implies* $\vdash^{\mathbf{CL}} x \doteq y, \binom{z}{x} \phi, \Gamma \Rightarrow \Delta$
where ϕ is a primitive formula.

where the condition of freshness of y means that y does not occur free in the contexts Γ, Δ nor is $y = x$.

Proof. See *Basic Proof Theory* by Troelstra and Schwitchenberg, Section 3.5 and Section 4.7. □

The proof system consisting of the axioms and proof rules of Lemma A.3.21 and Lemma A.3.22 is called *sequent calculus*. The fact that this is indeed a proof system is non-trivial, since to show that generalized transitivity holds it relies on the elimination of cuts (not further discussed here). In fact, as can be easily seen from the definition above, this proof system has a recursive relative demonstrability relation, an important result due to Gentzen. From this proof system it is also possible to derive the axioms and proof rules of Lemma A.3.16.

The classically provable formulas are formulas that can be proven using one of the proof systems described above.

Definition A.3.23. A theory T of first-order formulas is *deductively closed* if for every subset $\Gamma \subseteq T$ and first-order formula ϕ such that $\Gamma \vdash^{\mathbf{CL}} \phi$ also $\phi \in T$.

The notion of deductively closed can be generalized to other proof systems as well, and is not specific to **CL**.

Proposition A.3.24. *Every first-order theory $Th_1(\mathfrak{A})$ of a structure $\mathfrak{A}$ is deductively closed (with respect to* **CL***).*

A proof system is *finitary* whenever we have recursive enumerability of the provable formulas. Finitary proof systems are useful in practice, since they allow a computer to systematically generate proofs.

A.4 Soundness and completeness

The main results of first-order logic are now given. We assume Γ is a set of first-order formulas, and ϕ is a first-order formula.

Lemma A.4.1 (Soundness). $\Gamma \vdash^{\mathbf{CL}} \phi$ *implies* $\Gamma \models^{\mathbf{CL}} \phi$*.*

Lemma A.4.2 (Completeness). $\Gamma \models^{\mathbf{CL}} \phi$ *implies* $\Gamma \vdash^{\mathbf{CL}} \phi$*.*

Proof. Originally proven by Gödel [93]. See also the proof by Henkin [108], and Kleene's overview [136]. It can also be formally established using the interactive theorem provers Isabelle/HOL [191] and Coq [134]. □

Although we already established compactness of the satisfiability relation, in the sense of the compactness theorem (see Theorem A.2.10), we can now also show compactness of the semantic consequence relation in an alternative way. This follows easily from the fact that we have a finitary proof system that is sound and complete.

Theorem A.4.3. $\Gamma \models^{\mathbf{CL}} \phi$ *if and only if* $\Gamma_0 \models^{\mathbf{CL}} \phi$ *for some finite subset* $\Gamma_0 \subseteq \Gamma$*.*

Proof. Given that $\Gamma \models^{\mathbf{CL}} \phi$ holds, by completeness we have that $\Gamma \vdash^{\mathbf{CL}} \phi$. Since our proof system is finitary, there are only finitely many formulas in Γ used in the deduction of ϕ. Let Γ_0 be those formulas. Hence $\Gamma_0 \vdash^{\mathbf{CL}} \phi$, and by soundness $\Gamma_0 \models^{\mathbf{CL}} \phi$. The other direction is easy: given that $\Gamma_0 \models^{\mathbf{CL}} \phi$ for some finite subset $\Gamma_0 \subseteq \Gamma$, then we may always add more formulas to obtain $\Gamma \models^{\mathbf{CL}} \phi$. □

Theorem A.4.4 (Undecidability). *There is no algorithm that can decide whether* $\Gamma \models^{\mathbf{CL}} \phi$ *or not, for every signature, theory* Γ*, and formula* ϕ*.*

Proof. This is Church's theorem, see [23]. This result can also be established using Coq [120]. □

Note that the undecidability result is an existential statement: it does not mean there are no some signatures, theories, and formulas, for which the satisfiability relation is decidable. In fact, there are signatures and theories, for which the satisfiability relation is decidable.

A.5 Adding back terms

Up until now we have only considered signatures that consists of constant symbols, which are associated to a non-zero arity. In particular, signatures may contain predicate symbols (of arity 1) and relations symbols (of arity n for some $n > 1$) or constant symbols with third-order or higher-order arity. From a practical perspective, however, this set-up limits our ability to directly refer to individuals of the domain. Although we are able to give a name to individuals, e.g. in the context of a quantifier where an individual variable ranges over elements of the domain, and we are able to identify two individuals, we lack the ability to directly refer to individuals by some name, that denotes the individual regardless of the context in which that name appears.

In this section we consider an extension of the assertion language of classical logic in which we add facilities for referring to individuals of the domain directly. In our discussion we shall not formalize all aspects of our extension explicitly, leaving some details to the reader, and focus on first-order assertion languages. The purpose of our exposition is to show that this extension does not change the expressive power of first-order logic.

We have added $\doteq$ as a logical symbol, but consider for a moment an alternative approach: what if $\doteq$ is a 2-ary relation symbol? In second-order languages it is not needed to add such a relation to the signature, since the concept of identity is indirectly definable. The second-order sentence

$$\forall x, y.\ (x \doteq y) \leftrightarrow \forall P.\ P(x) \leftrightarrow P(y)$$

expresses that identity satisfies Leibniz's law of the *identity of indiscernibles* and its converse, the *indiscernability of identicals*: two elements share every property if and only if the two elements are identical. However, in first-order languages without identity as logical symbol, this cannot be expressed. Every structure gives an interpretation to the relation symbols, including $\doteq$ taken as relation symbol. A structure $\mathfrak{A} = (A, \mathcal{I})$ has a *standard interpretation of identity* if $\mathcal{I}$ assigns to the identity relation symbol $\doteq$ the value $\{(x, y) \mid x = y\} \subseteq A \times A$. Thus, in the standard interpretation, the extension of identity coincides with our meta-level concept of equality. This amounts to the same what we have accomplished in our definition of the satisfaction relation for the logical symbol $\doteq$.

Now we can introduce the derived concept of *unique existence.* We introduce the following abbreviation:

$$\exists! x\phi \text{ abbreviates } \exists x(\phi \wedge \forall y(\phi' \to x \doteq y))$$

where y is fresh (i.e. y is not x and does not occur in ϕ), and ϕ' is obtained from ϕ by renaming x to y. Semantically, we have that

$$\mathfrak{A}, \rho \models \exists! x\phi \text{ holds iff } \mathfrak{A}, \rho[x := a] \models \phi \text{ for a unique } a \in A.$$

The ability to express unique existence has two important consequences. Suppose that $\mathfrak{A} \models_\rho \exists! x\phi$ holds. Consider the case in which formula ϕ has only x as a free variable. By the coincidence condition, we then have that there must exists a unique $a \in A$ regardless of the valuation ρ. As such we say that ϕ *defines* the element a (in $\mathfrak{A}$). Similarly, consider the cases where all the free variables of formula ϕ are in the sequence of variables $x_1, \ldots, x_n, x$ (where we assume that all variables are distinct). Again by the coincidence condition we have that there must exists a unique total function $f : A^n \to A$ regardless of the valuation ρ:

$$f = \{(a_1, \ldots, a_n, a) \mid \mathfrak{A}, \rho[x_1 := a_1] \ldots [x_n := a_n][x := a] \models \phi\}$$

and we can again say that ϕ *defines* the total function f (in $\mathfrak{A}$).

Example A.5.1. Suppose we have a signature comprising the following constant symbols (and nothing else):

- we have the predicate symbol Z of arity 1,
- we have the relation symbol S of arity 2.

We can now state that there is *exactly* one element of the domain x such that $Z(x)$ holds by asserting

$$\exists! x Z(x).$$

Further, we can state that for every element of the domain x, there must be a *unique* y such that $S(x,y)$ holds, by asserting

$$\forall x \exists! y S(x,y).$$

However, can we also state that every element of the domain x is *reachable* either directly through Z, i.e. $Z(x)$ holds, or through a chain of S, i.e. $S(y,x)$ for some other reachable y? The following second-order sentence states this property:

$$\forall R.(\forall x.Z(x) \to R(x)) \wedge (\forall x,y.R(y) \wedge S(y,x) \to R(x)) \to \forall x R(x)$$

Note that not all structures (with a standard interpretation of identity) that satisfy the two first-order sentences also satisfy the second-order sentence.
End of Example.

A useful pattern emerges: if a predicate holds for exactly one element d of the domain, we can use that predicate as a way to identify element d. Intuitively, such a predicate identifies the element for which it holds. Similarly, if for a relation R, we have exactly one element of the domain d given elements $d_1, \ldots, d_{n-1}$ and $(d_1, \ldots, d_{n-1}, d) \in R$, then we can use that relation as a way to identify d once we are also able to identify the values of the first $n-1$ places. Taking both cases together, we say that a second-order constant symbol C of arity n has the property of *functionality* if

$$\forall x_1, \ldots, x_{n-1} \exists! x C(x_1, \ldots, x_{n-1}, x),$$

holds (if $n = 1$ the universal quantifiers are dropped).

We now extend our definition of signature in which we explicitly *declare* which constant symbols must have the above property of functionality. Predicate symbols of arity 1 that have the property of functionality are called *individual symbols* (or, more precisely, individual constant symbols). Relation symbols of arity n for $n > 1$ that have the property of functionality are called *function symbols.*

Remark A.5.2. We must make sure not to confuse constant symbols and individual (constant) symbols. Although individual symbols are constant symbols, the converse is not the case: all predicate symbols, relation symbols and function symbols are also constant symbols in the sense that their meaning does not depend on the context and remains fixed.

We fix a first-order signature Σ. As a syntactical convention, the uppercase constant symbols C are used for the constant symbols of our signature. If a constant symbol, say B, has been declared to have the property of functionality, then we use the lowercase symbol b instead. For individual constant symbols, we typically use the lowercase symbol c, and for function symbols we typically use the lowercase symbol f.

We now define terms and formulas and revisit some of the concepts introduced earlier. Note that we restrict ourselves to first-order assertion languages.

Definition A.5.3 (Terms)**.** A *term* of is constructed inductively as follows:

- any individual variable symbol x is a term,
- any individual constant symbol c is a term,
- given terms $t_1, \dots, t_n$ and function symbol f of arity $n+1$ then $f(t_1, \dots, t_n)$ is a term.

In the third clause, the terms $t_1, \dots, t_n$ are called the *arguments* of the function symbol f in the term $f(t_1, \dots, t_n)$. To be able to speak of the arity of a term, we let every term have the arity 0. We may see a term as a parse tree in which individual variable symbols or individual constant symbols appear at the leaves and function symbols at the branches.

Definition A.5.4 (Formulas)**.** A *formula* is constructed inductively as follows:

- $\bot$ is a formula,
- $P(t_1, \dots, t_n)$ is a formula if P is a constant symbol of arity n and $t_1, \dots, t_n$ are terms,
- $(\phi \to \psi)$ is a formula if ϕ and ψ are formulas,
- $(\forall x \phi)$ is a formula if ϕ is a formula and x an individual variable.

In the second clause, we do not restrict ourselves to only predicate symbols and relation symbols. All constant symbols (including those that have the property of functionality) are allowed at this level. This turns out to be useful when we show that extending first-order logic with terms does not increase the expressive power.

We can again define the set of free variables $FV(\phi)$ and bound variables $BV(\phi)$, but to do so we need to also introduce the set of free variables $FV(t)$ and bound variables $BV(t)$ for a given term t. The set of free variables $FV(t)$ consists precisely of all variables that occur in the term t, and the set of bound variables $BV(t)$ is empty. We can also extend variable renaming to terms, so that if π is a renaming of variable symbols (that preserves arity) we can define $\pi(t)$ by simultaneously replacing all the variable occurrences in t according to π.

A more general operation than variable renaming is the operation of substitution. Substitution is not simply a matter of replacing variables by terms. Rather, we define substitution as a transformation of formulas in two stages. This can be motivated by the following example. Given the formula $(\forall x.y = z)$, if we naively replace y by a term in which x occurs as a variable, then that variable now falls under the scope of the quantifier that binds x. The resulting formula thus has a term which may now have a different meaning than when considering that term in a different context, outside of the scope of the quantifier. That x would fall under the scope of the quantifier is called *variable capturing.* We intend to avoid variable capturing to ensure we can show the relation between (syntactic) substitutions and the semantics of terms, later on.

We also have substitutions of variables for terms that prevents variable capturing. Capture-avoiding substitution can be generalized to arbitrary formulas by first performing a renaming to obtain an alphabetic variant, for which the condition of the capture-avoiding substitution is always satisfied.

Definition A.5.5 (Capture-avoiding substitutions)**.** A substitution $\phi[x := t]$ is defined by replacing in ϕ all free occurrences of x by t, as long as the bound variables of ϕ are disjoint from the free variables of t.

It is now also possible to extend the definition of structures, so that the interpretation is extended to also interpret individual symbols and function symbols in a certain way to guarantee that the declared properties hold. For a given interpretation $\mathcal{I}$, we let $c^{\mathcal{I}}$ denote the interpretation of the constant symbol c, and $f^{\mathcal{I}}$ denote the interpretation of the function symbol f.

Definition A.5.6 (Evaluation function)**.** Given a structure $\mathfrak{A} = (A, \mathcal{I})$ and a valuation ρ of $\mathfrak{A}$, and a term t. The evaluation function $\mathfrak{A}[\![t]\!]^{\mathbf{CL}}_{\rho}$ is defined inductively on the structure of t:

- $\mathfrak{A}[\![x]\!]^{\mathbf{CL}}_{\rho} = \rho(x)$,
- $\mathfrak{A}[\![c]\!]^{\mathbf{CL}}_{\rho} = a$ where $a \in c^{\mathcal{I}}$,
- $\mathfrak{A}[\![f(t_1, \ldots, t_n)]\!]^{\mathbf{CL}}_{\rho} = a$ where $(\mathfrak{A}[\![t_1]\!]^{\mathbf{CL}}_{\rho}, \ldots, \mathfrak{A}[\![t_n]\!]^{\mathbf{CL}}_{\rho}, a) \in f^{\mathcal{I}}$.

We may drop the superscript **CL** if clear from context. In the second and third clause, we can pick *any* such a. Due to the restriction on the interpretation $\mathcal{I}$ on constant symbols which have the property of functionality, we know there exists a unique one. We may also simply write $\rho(t)$ for $\mathfrak{A}[\![t]\!]^{\mathbf{CL}}_{\rho}$, since every valuation is defined in a context where there is a known structure $\mathfrak{A}$.

It is also possible to redefine the satisfaction relation for formulas with terms, but we shall leave out the details. An important consequence of defining such satisfaction relation is the following lemma.

Lemma A.5.7 (Substitution lemma)**.**

$$\mathfrak{A}, \rho \models^{\mathbf{CL}} \phi[x := t] \textit{ if and only if } \mathfrak{A}, \rho[x := \rho(t)] \models^{\mathbf{CL}} \phi.$$

There is also a translation of formulas with terms to formulas without terms which has the same denotation, which allows us to eliminate all terms.

Bibliographic notes

C. Grabmayer's PhD thesis [97] contains a more detailed analysis of abstract proof systems and introduces the notions of derivability and admissibility of proof rules in an abstract setting [98].

Appendix B

Hoare's logic

Hoare's logic (some authors write: Hoare logic) was introduced by C.A.R. Hoare in 1969 [118], based on the inductive assertion method on flow charts as was introduced in 1967 by R.W. Floyd [84, 62]. See also the collection of fundamental papers on program verification by T.T.R. Colburn, J.H. Fetzer, and R.L. Rankin [54].

The main philosophical idea underlying both Hoare's logic and Floyd's inductive assertion method is that we have two 'modes' of description: that of *being*, and that of *change*. These two concepts are also understood as the *statics* and *dynamics* of a system, respectively. By making use of a logical language (e.g. first-order logic or separation logic) we can describe the state of being, by which one can think of a static snapshot in time of the currently realized memory state of a computer. By making use of a control structuring language (e.g. a flow chart or a program in a programming language), we can describe the change of state, which essentially describes a dynamic process which transforms initial states into final states or chains together such transformations. By combining descriptions of both modes, being and change, we obtain the program specification (also called the Hoare triple):

$$\{\phi\}\ S\ \{\psi\}$$

where ϕ is a description of the possible initial states (the precondition), S is a description of the program that transforms an initial state into a final state, and ψ is a description of the possible final states (the postcondition). A program S is correct with respect to a specification whenever it is indeed the case that, after executing the program from an initial state that satisfies the precondition, we obtain a state as prescribed by the postcondition. In some sense, a program specification describes the expected behavior of a program, whereas a program simply describes behavior.

Ideally there is no need for program specifications: by making sure a program exactly describes the change intended, it is always correct. Popularly, this is known by the phrase: "it is not a bug, it is a feature." However, the program specification adds redundancy to the program. The program describes *how* the state changes from an initial state into a final state, whereas the pre- and postconditions in a program specification describes logically *what* is the state of being before and after the execution of the program. The basic premise of program verification is that humans err—and by means of program specifications, where we combine two different languages in which one describes both the 'intended' program behavior and the 'actual' program behavior, we can detect errors if these two behaviors do not match.

Concretely, during execution of a program, there exists a current state of the values of memory. A basic program either performs a test, or performs an operation. A test is an inspection of the current state to check whether a condition on the state holds or not. An operation performs an action that may or may not change the state. By performing tests and operations, programs direct or control the flow of states, from an initial state to possibly a final state. Complex programs are composed out of programs with the intention to structure the flow of control, where tests can be used to influence the direction of flow of control. A processor is the component of a computer which, step by step, either performs a test or an operation, as specified by a program. The program thus is an input to the processor, and the program indirectly controls which tests and operations are performed by the processor. The result of the tests consequently direct the flow of control as described by a program, thus resulting in a feedback loop.

This conception of program originated in 1945, by the initial designs of John von Neumann's computer architecture [94]. Already in this early work, we can see that memory and program are separate, and we follow this design choice and also exclude so-called self-modifying programs. For practical reasons, we also see differences in the storage locations of memory: the current state of the memory of a computer can be divided into the internal state and the external state. The internal state stores the value of registers, the external state stores the value of addressable memory. The internal state is always directly accessible in tests, and can be operated on. The external state can be loaded and stored, being special operations. Thus, we do not have direct access to external state, only indirectly through load and store operations.

This division between internal and external state is more blurred in modern computer architectures of the past decades, by the introduction of cached memory—where parts of the external state is duplicated into hidden registers, which cannot be directly controlled by the program but mirror the behavior of the external memory. This architectural choice is mainly motivated by a further division of the use of the external memory into so-called stack memory and heap memory [111].

Since memory management is error-prone, high-level languages feature automatic memory management, while some high-level languages also still allow manual memory management. One form of automatic memory management is by using block scopes, in which a local variable is temporarily allocated on the stack and deallocated once the block has finished executing. Another form of automatic memory management is using garbage-collected heaps, in which all heap memory is scanned and regions of memory automatically deallocated if they no longer can influence the outcome of the execution of a program. We shall consider a programming language which features automatic stack memory management, but manual heap memory management—without garbage collection.

In this chapter we shall introduce the abstract syntax of programs, introduce three different styles of giving programs semantics (operational semantics, denotational semantics, and axiomatic semantics), and discuss the proof system for deriving correct program specifications called Hoare's logic. The ideas presented in Sections B.1, B.2, B.3 can be found in any competent book on program verification, such as [71, 61, 227, 100, 86, 10]. The use of program signatures and machine models, however, may be novel. The main motivation for revisiting the basic material is to present it in such a way to make it easy to adapt to separation logic, in Chapter 4. The material presented in Section B.5 is largely based on [29], but the presentation here is novel and the proof system more modular than in [29].

B.1 Syntax of programs

This section describes the syntax of a simplified programming language, necessary for supporting the semantics and proof system for reasoning about correctness of programs. Although we restrict ourselves to a simplified programming language, the expressivity of programs is nonetheless interesting: we include the Turing-complete programming languages.

When considering the syntax of programs, there is a design choice in formulating the programming language. One could give the concrete syntax of programs being particular constructions of statements, or one abstracts from the primitive operations and tests. In the latter case one can obtain the concrete syntax as a particular instance of the abstract syntax. This set-up of the syntax of programs is not much different than the set-up of formulas in the assertion language, where one separates the logical from the non-logical symbols by the introduction of a signature. Similarly, we could introduce the concept of a *program signature* which collects the primitive operations and tests out of which the statements are constructed.

Given a program signature that consists of the primitive operations and tests, we can form the statements of a program. Complex statements are constructed from compositions of simpler statements, recursively. One can represent statements by their parse trees in which at the leaves of the tree primitive operations and tests occur. One may compare statements of the programming language to formulas of the assertion language.

In fact, when one wants to extend our programming language to include recursive procedures, the abstract syntax approach is beneficial, since procedures can be considered particular primitive operations with a fixed interpretation given by a system of procedure declarations. This is not much different than having recursive predicates in the assertion language. We shall first focus on programs without recursive procedures, and can later add recursive procedures: in a sense, recursive procedures are an orthogonal concern.

Before introducing the formal definition of statements, we consider the possible effects that the executions of programs have on the states of a machine. We model these effects as a state transition, and the behavior of a program essentially is the possible sequences of state transitions. Each such sequence is also called an execution. An important design choice, however, is to contain the effect of primitive operations and tests, by limiting what part of the state an operation or test can (at most) be *accessed*, and what part of the state can (at most) be *changed* by an operation.

We reuse the concept of variables (see Definition A.1.1) to denote parts of the state. The accessible variables of a program restricts what part of the state can influence program behavior, and the changed variables of a program restricts what part of the state can be modified in the state transitions that constitute program behavior. In the context of a program we speak of *program variables*, whereas in the context of assertions we speak of *logical variables*. Although it is possible to also consider higher-order program variables, also called subscripted variables, we restrict ourselves to first-order program variables.

Given a program variable x, a *polarized variable* is either x or $\overline{x}$ (we say 'input x' or 'output x', respectively). The absence of the line indicates that the program variable is accessible, and the presence of the line indicates that the program variable is changed. Note that the presence of the line on top of a program variable makes it a different polarized variable, i.e. $x \neq \overline{x}$.

Definition B.1.1 (Program signature)**.** A *program signature* consists of a recursive set of *operations* and *tests*, such that each operation is associated with a finite set of polarized variables, and each test is associated with a finite set of (accessible) program variables.

We typically denote a program signature by Δ. If P is an operation of Δ, then we may speak of the accessible program variables $x_1, \ldots, x_n$ of P and the changed program variables $y_1, \ldots, y_m$ of P to mean that P is associated to the finite set $\{x_1, \ldots, x_n, \overline{y_1}, \ldots, \overline{y_m}\}$ of polarized variables. If T is a test of I, we write $T(x_1, \ldots, x_n)$ to mean that $x_1, \ldots, x_n$ are the accessible program variables associated to T.

Every first-order signature Σ induces a program signature, where all the tests are quantifier-free formulas. Although the tests are fixed, it still remains a design decision to select the appropriate operations: the selection of primitive operations thus affects what computations can be expressed by a program.

Definition B.1.2. A *first-order program signature* $FPS(\Sigma)$ is a program signature such that every test with accessible program variables $x_1, \ldots, x_n$ corresponds to a quantifier-free formula $\phi(x_1, \ldots, x_n)$, and also includes:

- the *assignment* operation $y := x$
  (where x is an accessible and y is a changed program variable).

The first-order program signature corresponds to register machines, where program variables are registers of the machine, and tests work on the registers of the machine. Note that the operations of the inherited program signature may access and change arbitrary program variables.

Block programs manipulate registers but may also temporarily store values by pushing them on the stack, and later retrieve old values by popping them from the stack. This is useful for implementing local variables, which are needed for introducing terms later on.

Definition B.1.3. A *block program signature* $BPS(\Sigma)$ is a first-order program signature $FPS(\Sigma)$ that also includes:

- the *parallel assignment* operation $\vec{y} := \vec{x}$
  (where $\vec{x} = x_1, \ldots, x_n$ are accessible and $\vec{y} = y_1, \ldots, y_n$ are changed),
- the *push* operation $\mathbf{push}(x)$
  (where x is an accessible program variable),
- the *pop* operation $\mathbf{pop}(x)$
  (where x is a changed program variable).

Note that for a push operation $\mathbf{push}(x)$ there are no changed variables. Although the stack is modified by this operation, the stack is left implicit (as an implementation detail) and as such not represented by a program variable.

Pointer programs not only manipulate values assigned to program variables but also values assigned to locations on the heap. As such, we consider an extension of first-order program signatures which includes operations for manipulating the heap. A pointer program signature also includes operations for looking up a value from the heap (lookup), modifying a value on the heap (mutation), allocating a new location with an initial value (allocation), and deallocating a location (deallocation).

Definition B.1.4. A *pointer program signature* $PPS(\Sigma)$ is a first-order program signature $FPS(\Sigma)$ that also includes:

- the *lookup* operation $x := [y]$
  (where y is an accessible and x is a changed program variable),
- the *mutation* operation $[x] := y$
  (where x and y are accessible program variables),
- the *allocation* operation $x := \mathbf{new}(y)$
  (where y is an accessible and x is a changed program variable),
- the *deallocation* operation $\mathbf{delete}(x)$
  (where x is an accessible program variable).

Note that for a mutation operation $[x] := y$ there are no changed variables. Although the heap is modified by this operation, the heap is implicit and as such not represented by a program variable. Also the lookup, allocation and deallocation operations above have a side-effect, namely they modify the implicit heap. This phenomenon, where no variables (or not all) are changed but there is a (hidden) state change, is in general called a *side-effect.*

Each program signature (including the standard, pointer and block program signatures) can be used to generate statements. We fix some program signature for the remainder of this section, unless explicitly mentioned otherwise.

Definition B.1.5 (Statements)**.** Given a program signature Δ. A statement is constructed inductively as follows:

1. O is a statement (called primitive operation) where O is an operation of Δ,
2. **skip** is a statement (called *no operation*),
3. **halt** is a statement (called *halt operation*),
4. $S_1; S_2$ is a statement (called *sequential composition*) given that S_1 and S_2 are statements,
5. **if** T **then** S_1 **else** S_2 **fi** is a statement (called *conditional* statement) given that T is a test of Δ, and S_1 and S_2 are statements,
6. **while** T **do** S **od** is a statement (called *looping* statement) given that T is a test of Δ and S is a statement.

All statements are constructed by one of these five clauses. Alternatively, we can define statements by the following abstract grammar:

$$S, S_1, S_2 ::= O \mid \mathbf{skip} \mid \mathbf{halt} \mid S_1; S_2 \mid \mathbf{if}\ T\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \mid \mathbf{while}\ T\ \mathbf{do}\ S\ \mathbf{od}.$$

The first two clauses construct primitive statements, the last three clauses construct complex statements. Sequential composition $S_1; S_2; S_3$ is ambiguous, but harmless as turns out later, and we may use parentheses around statements to disambiguate $(S_1; S_2); S_3$ and $S_1; (S_2; S_3)$. We also consider a *statement context*, denoted $S[-]$, which is a statement with exactly one hole in the place of a statement. The hole is denoted by $\Box$. Then $S[S_1]$ is a statement in which S_1 is plugged into the hole of the statement context.

The notion of accessible and changed variables can be lifted to statements S. We write $S(x_1, \ldots, x_n; y_1, \ldots, y_m)$ to mean that the accessible program variables of S are $x_1, \ldots, x_n$ and the changed program variables of S are $y_1, \ldots, y_m$. Sometimes it is easier to work with the finite sets of accessible and changed variables, denoted by *access*(S) and *change*(S), respectively. The accessible and changed variables of a statement are an over-approximation, in the sense that these are the *possible* accessed and changed variables. The set of program variables occurring in S is denoted *var*(S) and is the union of the accessible and changed variables.

Definition B.1.6 (Accessible and changed variables)**.** The accessible and changed program variables of a statement S is defined inductively on the structure of S:

- $\mathit{access}(O) = \{x_1, \ldots, x_n\}$ and $\mathit{change}(O) = \{y_1, \ldots, y_m\}$ given that the operation O is associated to the polarized variables $\{x_1, \ldots, x_n, \overline{y_1}, \ldots, \overline{y_m}\}$,
- $\mathit{access}(\mathbf{skip}) = \mathit{change}(\mathbf{skip}) = \emptyset$,
- $\mathit{access}(\mathbf{halt}) = \mathit{change}(\mathbf{halt}) = \emptyset$,
- $\mathit{access}(S_1; S_2) = \mathit{access}(S_1) \cup \mathit{access}(S_2)$,

- $\mathit{change}(S_1; S_2) = \mathit{change}(S_1) \cup \mathit{change}(S_2)$,
- $\mathit{access}(\textbf{if } T \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \mathit{access}(S_1) \cup \mathit{access}(S_2) \cup \{x_1, \ldots, x_n\}$ given $T(x_1, \ldots, x_n)$,
- $\mathit{change}(\textbf{if } T \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \mathit{change}(S_1) \cup \mathit{change}(S_2)$,
- $\mathit{access}(\textbf{while } T \textbf{ do } S \textbf{ od}) = \mathit{access}(S) \cup \{x_1, \ldots, x_n\}$ given $T(x_1, \ldots, x_n)$,
- $\mathit{change}(\textbf{while } T \textbf{ do } S \textbf{ od}) = \mathit{change}(S)$.

Note that the definitions of *access* and *change* do not depend on each other. The accessible program variables can be approximated more precisely by stating

$$\mathit{access}(S_1; S_2) = \mathit{access}(S_1) \cup (\mathit{access}(S_2) \setminus \mathit{change}(S_1))$$

where now the changed program variables of S_1 act as 'binders' with respect to the statement S_2. In fact,

$$\begin{aligned}
&\mathit{access}(S_1; (S_2; S_3)) \\
={}& \mathit{access}(S_1) \cup (\mathit{access}(S_2; S_3) \setminus \mathit{change}(S_1)) \\
={}& \mathit{access}(S_1) \cup ((\mathit{access}(S_2) \cup (\mathit{access}(S_3) \setminus \mathit{change}(S_2))) \setminus \mathit{change}(S_1)) \\
={}& \mathit{access}(S_1) \cup (\mathit{access}(S_2) \setminus \mathit{change}(S_1)) \cup (\mathit{access}(S_3) \setminus \mathit{change}(S_1; S_2)) \\
={}& \mathit{access}(S_1; S_2) \cup (\mathit{access}(S_3) \setminus \mathit{change}(S_1; S_2)) \\
={}& \mathit{access}((S_1; S_2); S_3)
\end{aligned}$$

However, this may complicate the proofs involving accessible and changed variables later on. Hence, we opt for the simpler definition given above, which is a cruder approximation of the accessible variables.

We could define complex tests by the following abstract grammar:

$$B, B_1, B_2 ::= T \mid \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2$$

where negation binds strongest, conjunction binds more strongly than disjunction, and the other ambiguities are harmless. We can then introduce abbreviations for statements:

$$\begin{aligned}
\textbf{if } \neg B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} &:= \textbf{if } B \textbf{ then } S_2 \textbf{ else } S_1 \textbf{ fi} \\
\textbf{if } B_1 \wedge B_2 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} &:= \textbf{if } B_1 \textbf{ then if } B_2 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi else } S_2 \textbf{ fi} \\
\textbf{if } B_1 \vee B_2 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} &:= \textbf{if } B_1 \textbf{ then } S_1 \textbf{ else if } B_2 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi fi} \\
\textbf{if } B \textbf{ then } S_1 \textbf{ fi} &:= \textbf{if } B \textbf{ then } S_1 \textbf{ else skip fi}
\end{aligned}$$

In the case of a first-order program signature, where tests are quantifier-free formulas, this may lead to an ambiguous interpretation of tests. However, it turns out that the semantics we give later on assigns the same meaning to both readings, so this ambiguity is harmless.

We can introduce assertions by the following abbreviation:

$$\textbf{assert}(B) := \textbf{if } B \textbf{ then skip else halt fi}.$$

B.2 Operational semantics

The first approach of giving semantics to programs is that of *operational semantics*. In our operational semantics, we introduce *machine models* that consists of a state space and an *operationalization* of the primitive operations and tests. The semantics of a program is understood as a sequence of steps, which are taken as the processor instructs the machine to perform primitive operations or tests. We here abstract from the particular machine by the means of a machine model, hence our operational semantics is too an abstraction of actual processor behavior.

This approach of abstractly giving semantics to programs is similar to giving semantics to formulas, in which we have introduced structures that consists of a domain of values and an interpretation of the non-logical symbols. A program denotes behavior relative to a given machine model and initial state, similar to how a formula denotes a truth value relative to a given structure and valuation.

Machine models can, for example, be used to show that program transformations preserve the semantics of programs. Low-level programming languages are used to instruct hardware to perform operations and tests, whereas high-level programming languages abstract away from intricate or irrelevant low-level details (such as ordering or encoding details). These programming languages have different program signatures and machine models. A compiler transforms high-level programs into low-level programs, and the preservation of behavior of the respective programs can be shown by relating machine models (e.g. the behavior relative to one machine model can be simulated by behavior of another machine model).

The syntactical structure by which statements are formed is chosen in such way that it is suitable for giving semantics to programs, which is based on the execution behavior of the statements of a program. This allows for structured programming, or *control-structured programming*, where it is possible to recognize from the syntax of a program what are the so-called *control points* for which it is possible to reason about the possible states of the underlying machine model. Making the state of the underlying machine model predictable is an important property of the semantics, which ensures that one is able to reason about program correctness.

The set of *control points* of a program can be understood as follows. One may consider a program to be the top-level statement containing sub-statements. Any position before or after a sub-statement of a program is a control point. The semantics of programs is given in terms of a current control point, which moves around the program as statements are executed, one after the other. The intuitive idea of control points are formalized by the *continuation* of a statement, that is, what remains to be executed after a small-step in the execution of a statement is taken. The continuation of a statement is either another statement, or the termination marker $\checkmark$.

It is important to realize that the models we consider are *abstractions* of the primitive operations and tests that can be performed on actual machines that execute programs. Based on a machine model, we can define a state transition system that *abstractly* models the behavior of programs. The accuracy of the analysis of program behavior thus relies on the precision of the chosen machine model.

Therefore, it is a design decision how much we are concerned about the possibility of blocking, non-determinism, and failure. The execution of primitive operations may result in a blocking (e.g. the machine hangs and cannot progress), in an un(der)specified next state (e.g. the next state is not fully determined by the previous state and the operation performed), or in an explicit failure (e.g. the machine signals an error). We introduce failure-sensitive machine models which take these possibilities into account.

Definition B.2.1 (Failure-sensitive machine model)**.** A *failure-sensitive machine model* $\mathcal{M}$ is a pair of a state space $\mathcal{S}$ (a set of states), and an *operationalization* consisting of:

- for each operation O, a *transition function* which is a partial function $O^{\mathcal{M}}$ of states to a set of states,
- for each test T, a set of states $T^{\mathcal{M}}$.

Given input state s and operation O, we write $s' \in O^{\mathcal{M}}(s)$ if $O^{\mathcal{M}}(s)$ is defined and s' is in the set $O^{\mathcal{M}}(s)$. In that case, s' is an output state. We write $O^{\mathcal{M}}(s) = \emptyset$ if $O^{\mathcal{M}}(s)$ is defined and is empty. We write $O^{\mathcal{M}}(s) = \textbf{fail}$ if $O^{\mathcal{M}}(s)$ is undefined.

For a complex test B, we also have the induced set of states $B^{\mathcal{M}}$ as follows: $T^{\mathcal{M}}$ as base case, $\neg B^{\mathcal{M}}$ is the complement of $B^{\mathcal{M}}$, $(B_1 \wedge B_2)^{\mathcal{M}}$ is the intersection of $B_1^{\mathcal{M}}$ and $B_2^{\mathcal{M}}$, and $(B_1 \vee B_2)^{\mathcal{M}}$ is the union of $B_1^{\mathcal{M}}$ and $B_2^{\mathcal{M}}$.

One may picture a failure-sensitive machine model by means of a graph, in which the states are vertices and directed edges, labeled by a primitive operation, represent transitions from one state to another. The graph also may have loose ends, which are outgoing edges from one state that leads to no state at all. The transition function of a machine model determines the edges (possibly with a loose end) that are present in the graph. We can then picture the following properties of the transition function $O^{\mathcal{M}}$ of a machine model $\mathcal{M}$:

- If $O^{\mathcal{M}}(s) = \emptyset$, then the state s is an *indeterminate state* or a *blocked state* (with respect to operation O), that is, the next state is implicitly undefined. This may be picture by having no outgoing edges from the state s. We may think of hanging at the indeterminate or blocked state, being unable to go to the next state. *Indeterminacy* of the machine model $\mathcal{M}$ means that there exists an indeterminate state, and a machine model $\mathcal{M}$ is *progressive* if there are no blocked states (so it is impossible to hang by performing an operation).
- If $O^{\mathcal{M}}(s) = \{s'\}$ for some s', then the state s is a *deterministic state* (with respect to operation O), that is, there is exactly one outgoing edge from s into the next state s'. If every state of machine model $\mathcal{M}$ is deterministic, then $\mathcal{M}$ is also called *deterministic.*
- If $s' \in O^{\mathcal{M}}(s)$ and $s'' \in O^{\mathcal{M}}(s)$ for different s' and s'', then the state s is a *non-deterministic state* (with respect to operation O), that is, there are multiple outgoing edges from s. We may think of the next state of s to be arbitrary selected from the non-empty set $O^{\mathcal{M}}(s)$. The machine model $\mathcal{M}$ is called *non-deterministic* if there is a non-deterministic state.

- If $O^{\mathcal{M}}(s) = \textbf{fail}$, then the state s is a *failing state* (with respect to operation O), that is, the next state is explicitly undefined. This may be pictured by a loose end in the graph. A machine model which has a failing state is called *failing*, and a machine model without any failing state is called *non-failing.*

To avoid confusion, we do not speak of the 'determinacy' of a machine model $\mathcal{M}$. Some authors use 'determinacy' to mean deterministic. However, 'determinacy' may also mean the lack of indeterminacy of a machine model, i.e. that performing an operation in every state leads to some next state or is a failing state.

Now consider the processor, which consists of a programmable controller and a machine which is being controlled. The controller takes a program that specifies what operations the machine must perform, and what tests of the machine influence the control flow. After the controller reaches the end of the program, the processor terminates. Given a failure-sensitive machine model, we can also model the behavior of programs as being executed step-by-step by an *abstract* processor. The processor is abstractly modeled using configurations and transitions between configurations.

Definition B.2.2 (Configuration)**.** Given a failure-sensitive machine model $\mathcal{M}$. A configuration is a pair of continuation and a state of $\mathcal{M}$, or a failure signal **fail**.

Given a statement S and state s, we thus have that (S, s) is a configuration. The configuration $(\checkmark, s)$ is called a *terminal configuration*, and **fail** is called the *failure configuration.*

There are different approaches for modeling processor behavior, from fine-grained to coarse-grained. In the fine-grained approach we transition from configuration to configuration in small steps, and this approach is also called *small-step operational semantics* where the intermediate configuration between initial and final configuration are taken into account. In the coarse-grained approach we transition from initial configuration directly to (one possible) final configuration. There, the initial configuration is not related to any intermediary configurations.

The small-step semantics is closer to the behavior of an actual processor, whereas the big-step semantics is easier to reason about and allows us to show important closure properties of the semantics, such as compositionality. We take both approaches, and, in fact, both approaches are equivalent in a certain sense.

Definition B.2.3 (Small-step operational semantics)**.** Given a machine model $\mathcal{M}$. We define the binary relation $\longrightarrow$ on configurations as the smallest relation satisfying the following conditions:

$$\begin{aligned}
(O, s) &\longrightarrow (\checkmark, s') \text{ if } s' \in O^{\mathcal{M}}(s)\\
(O, s) &\longrightarrow \textbf{fail} \text{ if } O^{\mathcal{M}}(s) = \textbf{fail}\\
(\textbf{skip}, s) &\longrightarrow (\checkmark, s)\\
(S_1; S_2, s) &\longrightarrow (S_1'; S_2, s') \text{ if } (S_1, s) \longrightarrow (S_1', s')\\
(S_1; S_2, s) &\longrightarrow (S_2, s') \text{ if } (S_1, s) \longrightarrow (\checkmark, s')\\
(S_1; S_2, s) &\longrightarrow \textbf{fail} \text{ if } (S_1, s) \longrightarrow \textbf{fail}
\end{aligned}$$

$$(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, s) \longrightarrow (S_1, s) \text{ if } s \in B^{\mathcal{M}}$$
$$(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, s) \longrightarrow (S_2, s) \text{ if } s \notin B^{\mathcal{M}}$$
$$(\textbf{while } B \textbf{ do } S \textbf{ od}, s) \longrightarrow (S; \textbf{while } B \textbf{ do } S \textbf{ od}, s) \text{ if } s \in B^{\mathcal{M}}$$
$$(\textbf{while } B \textbf{ do } S \textbf{ od}, s) \longrightarrow (\checkmark, s) \text{ if } s \notin B^{\mathcal{M}}$$

where s is a state and S_1, S_2, S are statements.

We write $(S, s) \not\longrightarrow$ if there is no configuration C such that $(S, s) \longrightarrow C$. We have $(\textbf{halt}, s) \not\longrightarrow$.

Proposition B.2.4. *If* $(S_1, s) \not\longrightarrow$ *then* $(S_1; S_2, s) \not\longrightarrow$.

Our intuition is that the small-step relation operates on a single statement at a time. The sub-statement S' of S on which we operate, is called the *primed statement* of S. For each statement S, we also have a *primed context* $S[-]'$. It is the case that $S = S[S']'$, meaning that the statement S is obtained from its primed context $S[-]'$ for which in the hole the primed statement S' is plugged. The primed statement of S is any statement that is not a sequential composition. For any statement that is not sequential composition, the primed statement is that statement itself and the primed context is just a single hole $\square$. For the sequential composition $S_1; S_2$, we have that the primed statement of $S_1; S_2$ is the primed statement of S_1 and the primed context is $S_1[-]'; S_2$ where $S_1[-]'$ is the primed context of S_1.

Proposition B.2.5. *Let S' be the primed statement of S. We have the following:*

- $(S, s) \longrightarrow (S[S'']', s')$ *if* $(S', s) \longrightarrow (S'', s')$,
- $(S, s) \longrightarrow \textbf{fail}$ *if* $(S', s) \longrightarrow \textbf{fail}$,
- $(S, s) \not\longrightarrow$ *if* $(S', s) \not\longrightarrow$.

Proof. We have $S = S[S']'$. By structural induction on S. If $S[-]'$ is just a hole, the result follows immediately. Otherwise, we apply the small-step semantics for sequential composition and the induction hypothesis. □

After execution of the primed statement S' of S is finished execution continues with the *remainder* of statement S, which we denote by $R(S)$. The remainder of a statement is a continuation, to take into account the possibility that the remainder is not a statement. For any statement that is not sequential composition, the remainder is $\checkmark$. For the sequential composition $S_1; S_2$, there are two cases. The remainder of $S_1; S_2$ is S_2 if the remainder of S_1 is $\checkmark$. The remainder of $S_1; S_2$ is $S_1'; S_2$ if the remainder of S_1 is S_1'.

Proposition B.2.6. *Let S' be the primed statement of S.* $(S, s) \longrightarrow (R(S), s')$ *if* $(S', s) \longrightarrow (\checkmark, s')$.

Proof. Again by structural induction on S: all cases except sequential composition are trivial. For the remaining case where $S = S_1; S_2$, we distinguish the two cases whether $R(S_1) = \checkmark$ or not, and apply the relevant small-step semantics for sequential composition. □

By the above proposition, we now have established that the small step semantics either takes a step directly at top-level for any statement that is not a sequential composition, or takes a step at the primed statement. Since the primed statement is never a sequential composition, the step taken at the primed statement is a top-level step as well.

In fact, we can say something stronger about the small-step semantics defined above.

Proposition B.2.7 (Determinism)**.** *Given a deterministic machine model $\mathcal{M}$, then the small-step operational semantics satisfies the following property:*

$$\textit{if } C_1 \longrightarrow C_2 \textit{ and } C_1 \longrightarrow C_3 \textit{ then } C_2 = C_3.$$

The above definition of the small-step operation semantics defines a relation between two configurations. We can imagine a chain of configurations as related by the small-step relation $\longrightarrow$:

$$C_1 \longrightarrow C_2 \longrightarrow \ldots \longrightarrow C_n \longrightarrow \ldots.$$

Definition B.2.8. An *execution* is a chain of configurations related by $\longrightarrow$, which is either finite or infinite.

A *complete execution* is a finite chain with no further step possible, that is, there is no C_{n+1} such that $C_n \longrightarrow C_{n+1}$. A complete execution is necessarily finite. A complete execution *leading to termination* is a complete execution that ends in some terminal configuration $(\checkmark, s)$. A complete execution *leading to failure* is a complete execution that ends in the failure configuration **fail**. A complete execution *leading nowhere* is a complete execution that is neither leading to termination, nor leading to failure. In particular a configuration $C = (S, s)$, where s is an indeterminate state with respect to O and O is a primed statement of S, is called an *indeterminate configuration*. An complete execution that ends in an indeterminate configuration leads to nowhere. Executions leading nowhere are also called stuck. A *diverging execution* is an execution with an infinite chain of configurations.

For a given execution, the first configuration is called the *initial configuration*. Conversely, an initial configuration I induces the set of complete or diverging executions that start in the configuration I. For a complete execution, the last configuration is called the *final configuration*. We may also speak of a *reachable configuration* C from an initial configuration I if there exists an execution that starts in configuration I which contains the configuration C in its chain. A configuration is *unreachable* from an initial configuration if it is not reachable.

We may imagine the set of all executions, of a given machine model, to form a forest: the roots are initial configurations which induce (possibly infinite) trees formed by executions that share the same initial configuration, and for which shared prefixes of two different executions forms the trunk of a tree. The trunk splits in two or more branches by taking a small-step of an operation on a state that is non-deterministic. There are three kinds of leaves. First, a leave represents that a small-step of an operation is performed on a blocking state: there is no next configuration. Second, a leave represents the failure configuration **fail**. Third, the leave represents a terminal configuration $(\checkmark, s)$. The height of a configuration in a tree represents the number of small steps taken from the initial configuration (and as such there are no cycles back to earlier configurations), and this height is often used to inductively reason about properties of executions.

The concepts of two executions reaching the same configuration is important enough that it warrants its own definition.

Definition B.2.9 (Computations)**.** Given an initial configuration I and a configuration C, the set of executions starting in I and reaching C is a *computation* from I to C.

A computation consists of zero or more executions, all reaching the same intermediary configuration. A computation thus abstracts from the particular way of reaching this configuration. A computation for which in the reached configuration C there is still a further step possible may also consists of diverging executions. If for the reached configuration C no further step is possible, we call the computation *complete*. A complete computation necessarily consists of complete executions. A complete computation *leading to termination* is a complete computation that ends in some terminal configuration $(\checkmark, s)$. A complete computation *leading to failure* is a complete computation that ends in the failure configuration **fail**. A complete computation *leading nowhere* is a complete computation that is neither leading to termination, nor leading to failure.

There may be different computations starting from the initial configuration. In fact, an initial configuration I also induces a set of computations that start in the configuration I. It may be the case that for the same initial configuration I, there is a complete computation but also a computation which consists of diverging executions.

We now introduce the big-step semantics, which captures the notion of complete computations leading to termination or failure directly.

Definition B.2.10 (Big-step operational semantics)**.** Given a machine model $\mathcal{M}$. We define the binary relation $\twoheadrightarrow$ on configurations as the smallest relation satisfying the following conditions:

$$\begin{aligned}
(O, s) &\twoheadrightarrow (\checkmark, s') \text{ if } s' \in O^{\mathcal{M}}(s)\\
(O, s) &\twoheadrightarrow \textbf{fail} \text{ if } O^{\mathcal{M}}(s) = \textbf{fail}\\
(\textbf{skip}, s) &\twoheadrightarrow (\checkmark, s)\\
(S_1; S_2, s) &\twoheadrightarrow (\checkmark, s'') \text{ if } (S_1, s) \twoheadrightarrow (\checkmark, s') \text{ and } (S_2, s') \twoheadrightarrow (\checkmark, s'')\\
(S_1; S_2, s) &\twoheadrightarrow \textbf{fail} \text{ if } (S_1, s) \twoheadrightarrow \textbf{fail}\\
(S_1; S_2, s) &\twoheadrightarrow \textbf{fail} \text{ if } (S_1, s) \twoheadrightarrow (\checkmark, s') \text{ and } (S_2, s') \twoheadrightarrow \textbf{fail}
\end{aligned}$$

$$\begin{aligned}
(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, s) &\twoheadrightarrow C \text{ if } (S_1, s) \twoheadrightarrow C \text{ and } s \in B^{\mathcal{M}} \\
(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, s) &\twoheadrightarrow C \text{ if } (S_2, s) \twoheadrightarrow C \text{ and } s \notin B^{\mathcal{M}} \\
(\textbf{while } B \textbf{ do } S \textbf{ od}, s) &\twoheadrightarrow C \text{ if } (S; \textbf{while } B \textbf{ do } S \textbf{ od}, s) \twoheadrightarrow C \text{ and } s \in B^{\mathcal{M}} \\
(\textbf{while } B \textbf{ do } S \textbf{ od}, s) &\twoheadrightarrow (\checkmark, s) \text{ if } s \notin T^{\mathcal{M}}
\end{aligned}$$

In the big-step semantics we can no longer distinguish between a diverging execution or an execution leading to nowhere. In both cases we have that, for an initial configuration I, there is no final configuration C such that $I \twoheadrightarrow C$. It is possible to observe this difference from the small-step semantics: for the initial configuration there is either a diverging execution, in the first case, or a complete execution $I \longrightarrow \ldots \longrightarrow C$ that is stuck in C, when there is no step possible from C, in the second case. However, this distinction disappears if we only consider observing final configurations of either the form $(\checkmark, s)$ or **fail**.

In fact, there is a correspondence between big-step and small-step semantics. Let $\longrightarrow^+$ be the transitive closure of the binary relation $\longrightarrow$. And let $I \longrightarrow^n C$ denote that C can be reached in exactly n small steps from I. We then have that $I \longrightarrow^+ C$ holds if and only if there exists $n > 0$ such that $I \longrightarrow^n C$.

Proposition B.2.11. *The following holds:*

1. $(S_1; S_2, s) \longrightarrow^+ (S_1'; S_2, s')$ *if* $(S_1, s) \longrightarrow^+ (S_1', s')$,
2. $(S_1; S_2, s) \longrightarrow^+ (S_2, s')$ *if* $(S_1, s) \longrightarrow^+ (\checkmark, s')$,
3. $(S_1; S_2, s) \longrightarrow^+ (\checkmark, s'')$ *if* $(S_1, s) \longrightarrow^+ (\checkmark, s')$ *and* $(S_2, s') \longrightarrow^+ (\checkmark, s'')$,
4. $(S_1; S_2, s) \longrightarrow^+ \textbf{fail}$ *if* $(S_1, s) \longrightarrow^+ \textbf{fail}$.

Proposition B.2.12. $C_1 \twoheadrightarrow C_3$ *if* $C_1 \longrightarrow C_2$ *and* $C_2 \twoheadrightarrow C_3$.

Lemma B.2.13 (Correspondence small-step and big-step semantics)**.**
$I \longrightarrow^+ C$ if and only if $I \twoheadrightarrow C$ for any terminal or failure configuration C.

Proof. I cannot be of the form $(\checkmark, s)$ or **fail** since both the small-step and big-step operational semantics do not have terminal or failure configurations on the left-hand side, while C must be of that form. So, we have that $I = (S, s)$ for some S, s, and $C = (\checkmark, s')$ for some s' or $C = \textbf{fail}$. ($\Longleftarrow$) By induction on the way $(S, s) \twoheadrightarrow (\checkmark, s')$ is established, transitivity of $\longrightarrow^+$, and Proposition B.2.11. ($\Longrightarrow$) Assume $(S, s) \longrightarrow^n (\checkmark, s')$ for some $n > 0$, we proceed by induction on n, and use Proposition B.2.12. □

Corollary B.2.14. *Given a deterministic machine model $\mathcal{M}$, then the big-step operational semantics satisfies the following property:*

$$\textit{if } C_1 \twoheadrightarrow C_2 \textit{ and } C_1 \twoheadrightarrow C_3 \textit{ then } C_2 = C_3.$$

Observe that the final configurations obtained by the big-step semantics are either $(\checkmark, s)$ or **fail**. Given an initial configuration, we have a *result set* of possible final configurations related to that initial configuration. The result set is either empty (divergence or stuck), or it contains the failure configuration **fail** alongside terminal configurations of the form $(\checkmark, s)$ for some state $s \in \mathcal{S}$. Note that, due to possible non-deterministic operationalizations of the machine model, the result set may contain at the same time different terminal configurations and the failure configuration. Each configuration in the result set is obtained by a different computation.

Instead of considering only computations that begin in an initial configuration with a fixed state, it is useful to abstract from the initial state. Thus, each statement S induces a result set that is indexed by an initial state $s \in \mathcal{S}$, i.e. $\{C \mid (S, s) \twoheadrightarrow C\}_{s\in\mathcal{S}}$. We can lift this construction to a *set* of initial states, and thereby also have that each statement induces a result set indexed by a set of initial states $X \subseteq \mathcal{S}$, i.e. $\{C \mid (S, s) \twoheadrightarrow C \text{ for some } s \in X\}_{X\subseteq\mathcal{S}}$.

We are interested in composing multiple results sets: given that a result set determines the possible outcomes of parts of a statement, can we compose result sets into the result set of the overall statement? To do so, we represent result sets more abstractly as an element of $\mathcal{P}(\mathcal{S} \uplus \{\mathbf{fail}\})$, viz. as a subset of the disjoint union of states and a failure marker. We could see the failure marker **fail** as an *improper state*, and every state $s \in \mathcal{S}$ as a *proper state*. Every statement induces a result set indexed by a (proper or improper) state. Equivalently, a statement induces a function $\mathcal{S} \uplus \{\mathbf{fail}\}$ to $\mathcal{P}(\mathcal{S} \uplus \{\mathbf{fail}\})$, which we denote by $\mathcal{M}[S]$, with the following specification:

$$\mathcal{M}[S](\mathbf{fail}) = \{\mathbf{fail}\},\ \mathcal{M}[S](s) = \{s' \mid (S, s) \twoheadrightarrow (\checkmark, s')\} \cup \{\mathbf{fail} \mid (S, s) \twoheadrightarrow \mathbf{fail}\}$$

and we can lift this function to result sets, being a set of (proper or improper) states $Y \subseteq \mathcal{S} \uplus \{\mathbf{fail}\}$, with the following specification:

$$\mathcal{M}[S](Y) = \bigcup_{y\in Y} \mathcal{M}[S](y)$$

where y is either the improper state **fail** or a proper state in $\mathcal{S}$. We thus obtain our desired form: every statement induces a result set indexed by a result set.

B.3 Denotational semantics

We now see a second approach of giving semantics to statements, called a *denotational semantics*. We can use functions on result sets as the *domain of denotation* of statements, and we work towards characterizing our denotation of statements in a syntax-directed manner. This allows for equational-style reasoning about the semantics of statements.

We first introduce an approximate denotation of a statement, and then define the denotation of a statement as the limit of the approximate denotation.

Definition B.3.1 (Denotational semantics)**.** Given a failure-sensitive machine model $\mathcal{M}$ and statement S. The *approximate denotation* of a statement $\mathcal{M}[\![S]\!]^n$ is a function on result sets, defined inductively on n and structurally on S as follows:

- $\mathcal{M}[\![O]\!]^n(Y) = (Y \cap \{\mathbf{fail}\}) \cup \bigcup\{O^{\mathcal{M}}(s)\}_{s\in(Y\cap\mathcal{S})}$,
- $\mathcal{M}[\![\mathbf{skip}]\!]^n = \mathrm{id}$,
- $\mathcal{M}[\![\mathbf{halt}]\!]^n(Y) = Y \cap \{\mathbf{fail}\}$,
- $\mathcal{M}[\![S_1; S_2]\!]^n = \mathcal{M}[\![S_2]\!]^n \circ \mathcal{M}[\![S_1]\!]^n$,
- $\mathcal{M}[\![\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}]\!]^n(Y) =$
  $(Y \cap \{\mathbf{fail}\}) \cup \mathcal{M}[\![S_1]\!]^n(Y \sqcap B^{\mathcal{M}}) \cup \mathcal{M}[\![S_1]\!]^n(Y \sqcap (\neg B)^{\mathcal{M}})$,
- $\mathcal{M}[\![\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}]\!]^0(Y) = Y \cap \{\mathbf{fail}\}$,
- $\mathcal{M}[\![\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}]\!]^{n+1} =$
  $\mathcal{M}[\![\mathbf{if}\ B\ \mathbf{then}\ S; (\mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od})\ \mathbf{fi}]\!]^n$,

where $Y \subseteq \mathcal{S} \uplus \{\mathbf{fail}\}$ is a result set, and $Y \sqcap X$ is the intersection of Y and a set of proper states X but which propagates failure, so $\mathbf{fail} \in (Y \sqcap X)$ if $\mathbf{fail} \in Y$. We define the *denotation* of a statement as the limit of the approximate denotation:

$$\mathcal{M}[\![S]\!](Y) = \bigcup_{n=0}^{\infty} \mathcal{M}[\![S]\!]^n(Y).$$

We may also write $\mathcal{M}[\![S]\!](s)$ to mean $\mathcal{M}[\![S]\!](\{s\})$ for a singleton proper state s. One may think of the parameter n as the maximal number of loop iterations for the outer **while**-statements, where the parameter decreases for statements that are directly nested under a **while**-statement. By using this parameter we ensure that the approximate denotational semantics is well-defined.

It is easy to see that the first three clauses of the approximate denotation also hold for the limit of the approximate denotation, that is:

- $\mathcal{M}[\![O]\!](Y) = (Y \cap \{\mathbf{fail}\}) \cup \bigcup\{O^{\mathcal{M}}(s)\}_{s\in(Y\cap\mathcal{S})}$,
- $\mathcal{M}[\![\mathbf{skip}]\!] = \mathrm{id}$,
- $\mathcal{M}[\![\mathbf{halt}]\!](Y) = Y \cap \{\mathbf{fail}\}$.

However, to establish that the other clauses also hold for the complex statements, we need to introduce further technical intermediary results. This is important to do, since it allows for syntax-directed reasoning about the limit of the approximate denotation

The approximate denotation given above has important properties, namely:

Lemma B.3.2 (Monotonicity)**.**

1. *Given result sets $X \subseteq Y$, $\mathcal{M}[\![S]\!]^n(X) \subseteq \mathcal{M}[\![S]\!]^n(Y)$ if $X \subseteq Y$.*
2. *Given a result set Y, $\mathcal{M}[\![S]\!]^n(Y) \subseteq \mathcal{M}[\![S]\!]^{n+1}(Y)$.*
3. *Given a result set Y and $n \leq m$, $\mathcal{M}[\![S]\!]^n(Y) \subseteq \mathcal{M}[\![S]\!]^m(Y)$.*

4. $\mathcal{M}[\![S_2]\!]^n \circ \mathcal{M}[\![S_1]\!]^m \subseteq \mathcal{M}[\![S_1; S_2]\!]^{\max(n,m)}$.

Proof. The third result depends on the second result, which depends on the first. The first two by induction on n and S, and the last by induction on m. The fourth result is by induction on n and S_2: the case analysis is delicate and non-trivial, and the other results are needed multiple times. □

It is easy to lift the first result, and have $\mathcal{M}[\![S]\!](X) \subseteq \mathcal{M}[\![S]\!](Y)$ if $X \subseteq Y$. The second and third results become trivial when we lift them.

Based on the properties of monotonicity we can prove the following properties of the (approximate) denotation. Recall that **assert**(B) abbreviates

$$\textbf{if } B \textbf{ then skip else halt fi}.$$

We have a few intermediary technical properties, needed to establish later results.

Proposition B.3.3.

1. $\mathcal{M}[\![S_1; (S_2; S_3)]\!]^n = \mathcal{M}[\![(S_1; S_2); S_3)]\!]^n$.
2. $\mathcal{M}[\![S]\!]^n = \mathcal{M}[\![\textbf{skip}; S]\!]^n$.
3. $\textbf{fail} \in X$ *implies* $\textbf{fail} \in \mathcal{M}[\![S]\!]^n(X)$.

These properties are easily lifted to the limit of the approximate denotation. We proceed to establish the following properties of the denotational semantics.

Lemma B.3.4.

- $\mathcal{M}[\![S_1; S_2]\!] = \mathcal{M}[\![S_2]\!] \circ \mathcal{M}[\![S_1]\!]$.
- $\mathcal{M}[\![\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!] = \mathcal{M}[\![\textbf{assert}(B); S_1]\!] \cup \mathcal{M}[\![\textbf{assert}(\neg B); S_2]\!]$.

To allow for syntax-directed reasoning about **while**-statements, we make use of syntactic approximations. We have the following recursively defined syntactic approximation called *loop unrolling*:

$$\begin{aligned}(\textbf{while } B \textbf{ do } S \textbf{ od})^0 &= \textbf{halt},\\ (\textbf{while } B \textbf{ do } S \textbf{ od})^{k+1} &= \textbf{if } B \textbf{ then } S; (\textbf{while } B \textbf{ do } S \textbf{ od})^k \textbf{ else skip fi}.\end{aligned}$$

The following holds for the denotational semantics.

Lemma B.3.5 (Loop unrolling)**.**

$$\mathcal{M}[\![\textbf{while } B \textbf{ do } S \textbf{ od}]\!] = \bigcup_{k=0}^{\infty} \mathcal{M}[\![(\textbf{while } B \textbf{ do } S \textbf{ od})^k]\!].$$

It is now easy to establish, for the denotational semantics, that we can replace statements by equivalent statements under any context. In other words, equivalent statements cannot be discriminated by any context. Let $S[-]$ be any statement with a hole.

Corollary B.3.6 (Compositionality). $\mathcal{M}[\![S[S_1]]\!] = \mathcal{M}[\![S[S_2]]\!]$ *if* $\mathcal{M}[\![S_1]\!] = \mathcal{M}[\![S_2]\!]$.

It is also possible to generalize compositionality to contexts with arbitrarily many holes, for which a similar property as above holds.

We can now establish the correspondence between the operational semantics and denotational semantics. This correspondence allows us to reason about the operational semantics of a program using the denotational semantics, and vice versa. Thus, both approaches in giving semantics coincide!

Lemma B.3.7 (Full abstraction). $\mathcal{M}[\![S]\!] = \mathcal{M}[S]$.

B.4 Axiomatic semantics

There is a third approach of giving semantics to statements, which is the *axiomatic semantics.* In this approach we express the behavior of programs in terms of program specifications. In contrast to the operational and denotational semantics which are relative to a *given* machine model and—so to speak—works 'from inside out', the axiomatic semantics consists of a set of program specifications and works 'from outside in'. In the axiomatic semantics we declare which expectations we have of the behavior of a program, without *a priori* knowing the inner workings of the primitive operations of a machine model.

The behavior of a program can be modeled by its *input/output behavior.* Operationally, one may see program behavior as a relation between input and output states, i.e. $t \in \mathcal{M}[S](s)$ indicates that output state t is related to the (singleton) input state s and **fail** $\in \mathcal{M}[S](s)$ indicates that the program leads to failure from the (singleton) input state s. Denotationally, we know that a statement S denotes a state transformer $\mathcal{M}[\![S]\!]$, and when X is a set of (proper or improper) states that $\mathcal{M}[\![S]\!](X)$ also is a set of (proper or improper) states. We have seen before that operational and denotation semantics coincide. However, this raises the question: what language can we use to describe states? We turn to that question later in this section.

First, we outline the usefulness of axiomatic semantics. Expected program behavior can be expressed by giving two descriptions: one of the set of input states, and one of the set of output states. Formally, the description of the input states is called a *precondition*, and the description of the output states is called a *postcondition.* We work directly with these descriptions by the use of *program specifications.* We introduce the notation $\{\phi\}\ S\ \{\psi\}$ for program specifications which consists of a precondition ϕ, the statement S, and the postcondition ψ. Then, given a suitable interpretation $[\![-]\!]$ of the precondition and postcondition as sets of (proper or improper) states, we can interpret program specifications as follows (called the *partial correctness* interpretation):

$$\mathcal{M} \models \{\phi\}\ S\ \{\psi\} \text{ if and only if } \mathcal{M}[\![S]\!]([\![\phi]\!]) \subseteq [\![\psi]\!].$$

By $\mathcal{M} \models \{\phi\}\ S\ \{\psi\}$ we then mean that the program specification $\{\phi\}\ S\ \{\psi\}$ is satisfied in the machine model $\mathcal{M}$. Next, we consider sets of program specifications: these are called *program theories.*

Similar how theories in (first-order or higher-order) logic can be used to classify structures, we can use program theories to classify machine models. By giving a set of program specifications which we *expect* to be satisfied in a given machine model, we constrain the possible choices of machine models that are possible. Note that such constraints can also be expressed for complex statements, e.g. involving control structures such as loops!

Similar how each structure induces a (first-order or higher-order) theory, we also have that each machine model induces a program theory. Given a machine model $\mathcal{M}$, then by $Th(\mathcal{M})$ we denote the set of program specifications that are satisfied in $\mathcal{M}$. It is then also possible to compare machine models by their induced program theories.

Example B.4.1. Say, we work with formulas of first-order logic for the pre- and postconditions, and we want to introduce a new complex programming construct which introduces *local program variables* with the syntax:

$$\textbf{begin local } x := y;\ S \textbf{ end}$$

The intended meaning is that the variable x is local to the execution of the statement S: it has an initial value determined by y, but the original value of x is restored after execution of S ends. To describe the semantics of such a programming construct using the axiomatic approach would amount to saying that the program theory must be closed under the following rule:

$$\frac{\{\phi\}\ x := y; S\ \{\psi\}}{\{\phi\}\ \textbf{begin local } x := y; S \textbf{ end}\ \{\psi\}} \text{ where } x \notin FV(\psi)$$

Thus we are able to *declaratively* specify properties of the semantics of this complex programming construct, without knowing how this construct decomposes into primitive operations nor saying anything about the underlying denotational or operational semantics. *End of Example.*

In the axiomatic approach of giving semantics, we intend to give semantics to programs in an abstract setting, without explicitly knowing the underlying (operational or denotational) semantics of the primitive operations. We thus need a language in which we can express the precondition and postcondition, to be able to describe the behavior of a program. In doing so, we have three desiderata of the language we choose:

1. The tests (and thus assertions) can be described by the language,
2. the language is expressive enough to describe the behavior of the primitive operations,
3. the language is closed under substitution, conjunction, and negation.

The first desideratum lets us speak of an *assertion language.* Although the second desideratum is necessary, often one is still free to choose an appropriate level of abstraction. Typically, one wishes to specify primitive operations in which the (concrete) implementation details are hidden, i.e. at the level of abstraction of the programming language itself. The third desideratum naturally leads us to choose a logical language.

We take as assertion language the language of one of the logics that we have introduced earlier. In doing so, we restrict the class of machine models to ensure that the set of states and the tests are compatible with the chosen logic. In this section we introduce *logical machine models*, corresponding to classical first-order logic. In logical machine models we take valuations as states and the (denotation of) quantifier-free formulas which are the tests. As such, we fix a first-order program signature $FPS(\Sigma)$, for a given first-order signature Σ.

Although we focus in this section on classical first-order logic, nothing prevents us from considering other classes of machine models to be associated to different logics. In fact, in later sections we also introduce the class of machine models corresponding to separation logic. In principle, one can choose any logic and make suitable design choices to map the chosen logic to a class of machine models. For practical purposes, one makes design choices in such a way as to ensure that assertions are decidable (i.e. tests can be effectively evaluated in any state) and primitive operations are computable. This motivates our choice above to consider the quantifier-free formulas as tests. In practice, one can also restrict the first-order signature Σ to ensure only a subset of the signature of the logic can be used in tests. When doing so, one may speak of the *logical signature* and the *program signature* that is a subset of the logical signature. For technical simplicity, we shall speak of only one signature.

Before giving logical machine models, we introduce the concept of equal sets of valuations modulo a set of variables. Let X and Y be sets of valuations of some structure $\mathfrak{A}$, and $Z \subseteq V$ be a set of variables. Then by $X \equiv Y$ **mod** Z we mean that the sets X and Y are in a correspondence such that for each valuation $\rho \in X$ that corresponds to a valuation $\rho' \in Y$ we have that $\rho[V \setminus Z] = \rho'[V \setminus Z]$. This notion is also defined when a function is applied on both sets that are in correspondence, and then takes the original correspondence. Further, by $X \equiv f(X)$ **mod** Z we mean that for each valuation $\rho \in X$ and valuation $\rho' \in f(\rho)$ we have that $\rho[V \setminus Z] = \rho'[V \setminus Z]$. These notions can be lifted in the obvious way to result sets (being sets consisting of valuations or **fail**), or sets of proper or improper states (being the disjoint union of proper states and the **fail** marker).

Definition B.4.2. A *logical machine model* $\mathcal{M}$ is a pair of a structure $\mathfrak{A}$ and an operationalization consisting of:

- for each operation O, a transition function which is a partial function $O^{\mathcal{M}}$ of valuations to a set of valuations of $\mathfrak{A}$,
- for every operation $x := y$, the transition function $(x := y)^{\mathcal{M}}$ is defined by mapping ρ to $\rho[x := \rho(y)]$,
- for the transition function $O^{\mathcal{M}}$ we have the *change condition* that either $O^{\mathcal{M}}(\rho) =$ **fail** or $\rho'[V_1 \setminus \mathit{change}(O)] = \rho[V_1 \setminus \mathit{change}(O)]$ for every $\rho' \in O^{\mathcal{M}}(\rho)$,

- for the transition function $O^{\mathcal{M}}$ we have the *access condition* that states that $O^{\mathcal{M}}(\rho) \equiv O^{\mathcal{M}}(\rho')$ **mod** $\mathit{var}(O)$ for every ρ, ρ' for which $\rho[\mathit{access}(O)] = \rho'[\mathit{access}(O)]$ holds.

The operations $x := y$ have a fixed operationalization, namely by assigning the value of y to the program variable x. Further, the change and access conditions can be explained as follows. For every operation, we require that only the changed program variables are actually modified by the operation. We require that only the accessible program variables can have an influence on the outcome of an operation. In fact, both conditions imply that operations depend only on finitely many variables and can affect only finitely many variables.

We also write $\langle \mathcal{M}, \mathfrak{A} \rangle$ for a logical machine model to indicate its underlying structure $\mathfrak{A}$. A logical machine model is a failure-sensitive machine model in the following sense: the state space of a logical machine model is the set of valuations of $\mathfrak{A}$, and the given operationalization induces an operationalization for tests by associating every quantifier-free formula ϕ to the set $\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$ that denotes the valuations that satisfy ϕ in structure $\mathfrak{A}$.

The access and change conditions can be lifted to statements S.

Lemma B.4.3 (Change Lemma). *Given a set of proper states X,*

$$X \equiv \langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](X) \ \mathbf{mod}\ \mathit{change}(S).$$

Lemma B.4.4 (Access Lemma). *Given two sets of proper states X, Y such that $X \equiv Y$ **mod** $(V \setminus \mathit{access}(S))$, then*

$$\langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](X) \equiv \langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](Y) \ \mathbf{mod}\ (V \setminus \mathit{var}(S)).$$

Intuitively, these express that a program only modifies the variables $\mathit{change}(S)$, and that the outcome of a program is only dependent on the variables $\mathit{access}(S)$.

We now formally define whether a program specification is satisfied in a logical machine model. Note that, contrary to our earlier discussion, there is a mismatch in the denotation of formulas and the sets of (proper or improper) states: formulas never denote the improper state **fail**. Thus we have a stronger interpretation for program specifications, called *strong partial correctness*, defined as such:

$$\langle \mathcal{M}, \mathfrak{A} \rangle \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\} \text{ if and only if } \langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}) \subseteq \mathfrak{A}[\![\psi]\!]^{\mathbf{CL}}.$$

Since **fail** is never in $\mathfrak{A}[\![\psi]\!]^{\mathbf{CL}}$, this interpretation explicitly states that the machine never fails when executing program S starting from any state in $\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$. Note that the superscript **HL** (short for Hoare's Logic) is used to be able to distinguish this interpretation from the one introduced in the next chapter, but may be dropped if it is clear from context what interpretation is intended.

It is useful to restrict our attention to particular logical machine models, that are based on a structure that satisfies a particular theory. There are two levels at which we recognize theories: *background theories* and *program theories*. Let T be a set of first-order formulas, called a *background theory*. We write $\models^{\mathbf{HL}}_T \{\phi\}\ S\ \{\psi\}$ to mean $\langle \mathcal{M}, \mathfrak{A} \rangle \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$ for every logical machine model $\langle \mathcal{M}, \mathfrak{A} \rangle$ such that

$\mathfrak{A} \models^{\mathbf{CL}} T$. We then say that the program specification is *valid* relative to the given background theory. We write $\models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$ if the program specification is valid in every logical machine model, regardless of background theory, and call it *universally valid.*

Let Γ be a set of program specifications, called a *program theory.* We write $\Gamma \models^{\mathbf{HL}}_T \{\phi\}\ S\ \{\psi\}$ to mean $\langle \mathcal{M}, \mathfrak{A}\rangle \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$ for every structure $\mathfrak{A}$ such that $\mathfrak{A} \models^{\mathbf{CL}} T$ and every logical machine model $\langle \mathcal{M}, \mathfrak{A}\rangle$ such that $\langle \mathcal{M}, \mathfrak{A}\rangle \models^{\mathbf{HL}} \{\phi'\}\ S'\ \{\psi'\}$ for each $\{\phi'\}\ S'\ \{\psi'\} \in \Gamma$. We then say that the program specification $\{\phi\}\ S\ \{\psi\}$ is a *semantic consequence of* Γ *with respect to the background theory* T. The notion $\Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$, that $\{\phi\}\ S\ \{\psi\}$ is a *semantic consequence* of Γ, can be defined (regardless of the background theory) in a similar way.

It is sufficient to focus on the semantic consequence relation, regardless of the background theory, by the following argument. Observe that the program specification $\{\top\}$ **skip** $\{\phi\}$ is satisfied in a logical machine model $\langle \mathcal{M}, \mathfrak{A}\rangle$ if and only if $\langle \mathcal{M}, \mathfrak{A}\rangle[\![\mathbf{skip}]\!](\mathfrak{A}[\![\top]\!]^{\mathbf{CL}}) \subseteq \mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$ if and only if $\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$ is the set of all proper states if and only if $\mathfrak{A} \models^{\mathbf{CL}} \phi$. Hence, every formula ϕ in the background theory can be represented by the program specification $\{\top\}$ **skip** $\{\phi\}$. Let T be a background theory, and T' be the corresponding set of program specifications in which we represent each formula $\phi \in T$ as a program specification $\{\top\}$ **skip** $\{\phi\} \in T'$. Then we have $\Gamma \models^{\mathbf{HL}}_T \{\phi\}\ S\ \{\psi\}$ if and only if $\Gamma \cup T' \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$. For notational convenience, we simply work with formulas instead of their representations as program specifications. Hence, we merge the notions of background theory and program theory, and simply speak of a *theory*, being a set of program specifications or formulas. Every theory has projections to its underlying background theory and program theory.

We introduce a proof system in which program specifications can be deduced. The purpose of using a proof system is that we can effectively check the deduction of program specifications. The proof system is set-up as a proof system with premises. Typically, we take the background theory as premises, from which we can derive program specifications. Our proof system is called *Hoare's logic*, or **HL** in short, in honor of C.A.R. Hoare (but, as mentioned in Section 1.4, the proof system given below is by K.R. Apt and F.S. de Boer).

Definition B.4.5. The proof system **HL** consists of:

- program specifications or formulas of first-order logic as objects,
- the smallest deduction relation $\vdash^{\mathbf{HL}}$ satisfying the conditions:

  (skip) $\vdash^{\mathbf{HL}} \{\phi\}$ **skip** $\{\phi\}$,

  (halt) $\vdash^{\mathbf{HL}} \{\phi\}$ **halt** $\{$**false**$\}$,

  (assign) $\vdash^{\mathbf{HL}} \{\phi[x := y]\}\ x := y\ \{\phi\}$,

  (comp) $\{\phi\}\ S_1\ \{\psi\}, \{\psi\}\ S_2\ \{\chi\} \vdash^{\mathbf{HL}} \{\phi\}\ S_1; S_2\ \{\chi\}$,

  (if) $\{\phi \wedge \chi\}\ S_1\ \{\psi\}, \{\phi \wedge \neg\chi\}\ S_2\ \{\psi\} \vdash^{\mathbf{HL}} \{\phi\}$ **if** χ **then** S_1 **else** S_2 **fi** $\{\psi\}$,

  (while) $\{\phi \wedge \chi\}\ S\ \{\phi\} \vdash^{\mathbf{HL}} \{\phi\}$ **while** χ **do** S **od** $\{\phi \wedge \neg\chi\}$,

(conseq) $(\phi' \to \phi), \{\phi\}\ S\ \{\psi\}, (\psi \to \psi') \vdash^{\mathbf{HL}} \{\phi'\}\ S\ \{\psi'\}$,

(subst) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{HL}} \{\phi[x := y]\}\ S\ \{\psi[x := y]\}$ for $x \notin \mathit{var}(S), y \notin \mathit{change}(S)$,

(invar) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{HL}} \{\phi \wedge \chi\}\ S\ \{\psi \wedge \chi\}$ if $FV(\chi) \cap \mathit{change}(S) = \emptyset$,

(∃-intro) $\{\phi\}\ S\ \{\psi\} \vdash^{\mathbf{HL}} \{\exists x \phi\}\ S\ \{\psi\}$ for $x \notin \mathit{var}(S) \cup FV(\psi)$.

Note how only the consequence proof rule (conseq) uses formulas as premises. Every deduction is a proof tree constructed in the usual way. Hence, a deduction has only finitely many premises, being either formulas or program specifications.

Let Γ be a given theory. We can now formulate that the proof system satisfies the following meta-theoretical property, relating the proof system to the semantic consequence relation on program specifications.

Lemma B.4.6 (Soundness).

$$\Gamma \vdash^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\} \textit{ implies } \Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}.$$

Proof. Generalized reflexivity and generalized transitivity holds for the strong partial correctness interpretation too, by induction on the structure of the deduction. We then verify the axioms: For (skip), we have $\models \{\phi\}$ **skip** $\{\phi\}$ regardless of Γ, and this easily follows from the denotational semantics. Also for (halt), we have $\models \{\phi\}$ **halt** $\{$**false**$\}$ regardless of Γ, similar to skip. For (assign), every logical machine model has a fixed interpretation of $x := y$, and thus we have the result by the substitution lemma. We have that (comp) follows directly from the denotational semantics. For the proof rule (if) we can perform a case distinction on whether χ holds or not, and from $\{\phi \wedge \chi\}\ S\ \{\psi\}$ we can obtain $\{\phi\}$ **assert**$(\chi); S\ \{\psi\}$ and similar for the other case. For the proof rule (while) we can do loop unrolling, and then observe that

$$\bigcup_{k=0}^{\infty} \langle \mathcal{M}, \mathfrak{A} \rangle [\![(\textbf{while } \chi \textbf{ do } S \textbf{ od})^k]\!] (\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}) \subseteq \mathfrak{A}[\![\phi \wedge \neg\chi]\!]^{\mathbf{CL}}$$

holds by considering that the following holds

$$\langle \mathcal{M}, \mathfrak{A} \rangle [\![(\textbf{while } \chi \textbf{ do } S \textbf{ od})^k]\!] (\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}) \subseteq \mathfrak{A}[\![\phi]\!]^{\mathbf{CL}}$$

for every k; the latter can be shown by induction on k and the premise. For the proof rule (conseq) the result holds for logical machine models in which the premises are satisfied.

The remaining rules can be proven sound, given the following intuition:

- In the substitution rule (subst) we make use of the access lemma to take any computation from $\{\phi\}\ S\ \{\psi\}$ and change the initial state with respect to variable x that is not occurring in S to obtain another computation (the variable x can then not be overwritten by S). The value to assign to x is the value of y, which must have the same value in the initial and final state due to the change lemma. The specification then is satisfied by applying the substitution lemma on the initial and final state.

- The invariance rule (invar) follows from the change lemma, where the denotation of χ depends entirely on its free variables, which values cannot change.
- The $\exists$-introduction rule ($\exists$-intro) follows from the access lemma, since the value of x cannot have any effect on the computation of S nor determine the denotation of ψ. □

In fact, under suitable assumptions of the expressivity of the assertion language, the converse can be stated as well. To do so, we introduce the notions of a *weakest precondition* and *strongest postcondition*, relative to a given theory Γ.

Remark B.4.7. Since we have limited the variables that are changed, and every program depends only on finitely many variables, it is possible to express the weakest precondition (strongest postcondition) by a formula. These conditions cannot be described by a (finite) formula if, for example, a primitive operation would affect the value of infinitely many variables, or would depend on the value of infinitely many variables. *End of Remark.*

Given a program S and formula ψ, let $\mathit{WP}_\Gamma(S, \psi)$ denote the weakest (liberal) precondition, a formula with the following properties:

- $\Gamma \models^{\mathbf{HL}} \{\mathit{WP}_\Gamma(S,\psi)\}\ S\ \{\psi\}$,
- $\Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$ implies $\Gamma \models \phi \to \mathit{WP}_\Gamma(S,\psi)$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(\mathbf{skip},\psi) \to \psi$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(\mathbf{halt},\psi) \to \bot$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(x := y,\psi) \to \psi[x := y]$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(S_1; S_2,\psi) \to \mathit{WP}_\Gamma(S_1, \mathit{WP}_\Gamma(S_2,\psi))$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(\mathbf{if}\ \chi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi},\psi) \to$
  $(\mathit{WP}_\Gamma(S_1,\psi) \wedge \chi) \vee (\mathit{WP}_\Gamma(S_2,\psi) \wedge \neg\chi)$,
- $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(\mathbf{while}\ \chi\ \mathbf{do}\ S\ \mathbf{od},\psi) \to$
  $(\chi \to \mathit{WP}_\Gamma(S, \mathit{WP}_\Gamma(\mathbf{while}\ \chi\ \mathbf{do}\ S\ \mathbf{od},\psi))) \wedge (\neg\chi \to \psi)$.

Note that in some of these conditions we use formulas ϕ, such that $\Gamma \models \phi$ means that for every logical machine model $\langle \mathcal{M}, \mathfrak{A}\rangle$ that satisfies the theory Γ, we must have that the formula ϕ is valid, that is, $\mathfrak{A} \models \phi$. Whether a formula exists, that can express the weakest precondition, is a property of the assertion language and the given theory: not all choices of signatures and theories allow us to express such weakest precondition as a formula.

There are some general properties that hold of the weakest precondition, as defined above, that show that the weakest precondition is closely related to our denotational semantics:

Proposition B.4.8. $\Gamma \models^{\mathbf{HL}} \mathit{WP}_\Gamma(S_1, \mathit{WP}_\Gamma(S_2,\psi)) \to \mathit{WP}_\Gamma(S_1; S_2,\psi)$.

Proof. By definition of the weakest precondition, we have

$$\Gamma \models^{\mathbf{HL}} \{\, WP_\Gamma(S_2, \psi)\}\ S_2\ \{\psi\}$$

and

$$\Gamma \models^{\mathbf{HL}} \{\, WP_\Gamma(S_1,\, WP_\Gamma(S_2, \psi))\}\ S_1\ \{\, WP_\Gamma(S_2, \psi)\}.$$

By the soundness of the composition rule, we thus have

$$\Gamma \models^{\mathbf{HL}} \{\, WP_\Gamma(S_1,\, WP_\Gamma(S_2, \psi))\}\ S_1; S_2\ \{\psi\}.$$

But from this, it follows that $\Gamma \models^{\mathbf{HL}} WP_\Gamma(S_1,\, WP_\Gamma(S_2, \psi)) \to WP_\Gamma(S_1; S_2, \psi)$ also from the definition of weakest precondition. □

Other converses of the other properties of the weakest precondition given above can be shown too: this establishes that we deal with equivalence and not merely logical implications.

In fact, for a given logical machine model $\langle \mathcal{M}, \mathfrak{A}\rangle$ we can precisely specify what the weakest precondition denotes. A logical machine model fixes the background theory $\mathit{Th}_1(\mathfrak{A})$ by the choice of the underlying structure $\mathfrak{A}$, and furthermore fixes the program theory $\mathit{Th}(\langle \mathcal{M}, \mathfrak{A}\rangle)$ by the operationalization of $\mathcal{M}$. So we can take as theory $\Gamma = \mathit{Th}_1(\mathfrak{A}) \cup \mathit{Th}(\langle \mathcal{M}, \mathfrak{A}\rangle)$. In this case we can simply speak of $WP(S, \phi)$, dropping the subscript and instead take the theory induced by the given model. Then, the weakest precondition is understood, semantically, to denote

$$\mathfrak{A}[\![\, WP(S, \psi)]\!]^{\mathbf{CL}} = \{\rho \mid \langle \mathcal{M}, \mathfrak{A}\rangle[\![S]\!](\rho) \subseteq \mathfrak{A}[\![\psi]\!]^{\mathbf{CL}}\}$$

where ρ ranges over proper states (the valuations of $\mathfrak{A}$). Since our semantics denotes the empty set for diverging programs, and the empty set is always included in any set of proper states, we thus have a weakest *liberal* precondition in the sense that we do not care about diverging computations.[1] Note that in this setting, we are able to distinguish a primitive operation leading to failure from an indeterminate primitive operation: in the former case the weakest precondition is empty (since **fail** is never contained in any set of proper states), whereas in the latter case the weakest precondition is the set of all proper states (since the empty set is always contained in any set of proper states).

Next, let $\mathit{NF}_\Gamma(S)$ denote a formula expressing the precondition so that computations of S do not lead to failure, and let $\mathit{SP}_\Gamma(\phi, S)$ denote a formula expressing the strongest postcondition, with the following properties:

- $\Gamma \models^{\mathbf{HL}} \{\mathit{NF}_\Gamma(S) \wedge \phi\}\ S\ \{\mathit{SP}_\Gamma(\phi, S)\}$,
- $\Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$ implies $\Gamma \models \mathit{SP}_\Gamma(\phi, S) \to \psi$.

Note the asymmetry between the weakest precondition and the strongest postcondition due to the failure-sensitive semantics: the strongest postcondition is only

[1] If *liberal* is not caring about getting stuck without making any progress, then *progressive* is caring about making progress. But 'weakest progressive precondition' is terminology I invented.

given in the case the precondition excludes the possibility any computation leads to failure.

Again, for a given logical machine model $\langle \mathcal{M}, \mathfrak{A} \rangle$ we can precisely specify what the non-failing formula and the strongest postcondition denotes. Since a particular logical machine model induces a particular theory, we simply write $\mathit{NF}(S)$ and $\mathit{SP}(\phi, S)$, dropping the subscript. We can take

$$\mathfrak{A}[\![\mathit{NF}(S)]\!]^{\mathbf{CL}} = \{\rho \mid \mathbf{fail} \notin \langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](\rho)\}$$

and note that, for deterministic machine models, $\mathit{NF}(S)$ and $\mathit{WP}(S, \mathbf{true})$ denote the same set of proper states. We also can take

$$\mathfrak{A}[\![\mathit{SP}(\phi, S)]\!]^{\mathbf{CL}} = \{\rho \mid \rho \in \langle \mathcal{M}, \mathfrak{A} \rangle [\![S]\!](\mathfrak{A}[\![\phi]\!]^{\mathbf{CL}})\}$$

where we take all proper final states starting from a state in the given precondition. Note that, for a non-deterministic program S, we may have that a particular given ϕ possibly leads to failure but in a non-deterministic way. In that case, $\mathit{NF}(S)$ is empty, since the failure cannot be avoided. However, $\mathit{SP}(\phi, S)$ then could still be non-empty for the computations that do not lead to failure, whereas $\mathit{SP}(\mathbf{false}, S)$ is empty. Hence there is a difference between $\mathit{SP}(\phi, S)$ and $\mathit{SP}(\mathit{NF}(S) \wedge \phi, S)$.

We now show the relative completeness result, by using the weakest (liberal) precondition. This completeness result is called *relative* since we have two assumptions: we assume the *expressivity* of the weakest precondition, and we assume that every logical truth is contained in the theory. The latter assumption is quite strong and may go beyond what is computable or even recursively enumerable, and hence we refer to that latter assumption as if we have access to an *oracle*.

Lemma B.4.9 (Relative completeness)**.** *Given a theory Γ where the weakest liberal precondition $\mathit{WP}_\Gamma(S, \psi)$ is expressible and Γ is maximally consistent (with respect to the background theory), then*

$$\Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\} \text{ implies } \Gamma \vdash^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$$

for all formulas ϕ, ψ.

Proof. We assume $\Gamma \models^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$. The proof goes as follows: it suffices to show that $\Gamma \vdash^{\mathbf{HL}} \{\mathit{WP}_\Gamma(S, \psi)\}\ S\ \{\psi\}$, since we obtain the desired result by an application of the consequence rule and the property that $\Gamma \models^{\mathbf{HL}} \phi \to \mathit{WP}_\Gamma(S, \psi)$. Since Γ is maximally consistent we can actually apply the consequence rule.

The proof is by induction on S. For primitive operations, the specification must follow from Γ (otherwise it contradicts our assumption). The skip, halt, and assignment operations follow from the properties of the weakest precondition.

For sequential composition, we apply the consequence rule (together with the property of the weakest precondition that distributes over composition) and need to show $\Gamma \vdash^{\mathbf{HL}} \{\mathit{WP}_\Gamma(S_1, \mathit{WP}_\Gamma(S_2, \psi))\}\ S_1; S_2\ \{\psi\}$. This can be done by an application of the composition rule, with $\mathit{WP}_\Gamma(S_2, \psi)$ as intermediary formula, and the induction hypotheses. The other complex statements are similar. □

Summarizing, the completeness result depends essentially on the expressivity of the weakest precondition. That this is crucial boils down to the following observation: if we know that $\Gamma \models^{\mathbf{HL}} \{\phi\}\ S_1; S_2\ \{\psi\}$, how can we find a description of the possible intermediate states? The weakest precondition offers such a description. Similarly, the weakest precondition describes the loop invariant in the case of the **while**-statement. The fact that we need an oracle is secondary. In the case we deal with finite structures, for which the background theory is complete, we also have completeness of Hoare's logic. For example, in the case of 32-bit signed integers, the oracle can be effectively implemented by means of a decision procedure, and the question whether a program specification is deducible is decidable as well. However, there are background theories such as the theory of stacks, for which one can give concrete programs where the loop invariant is non-expressible [133]. In that case, having an oracle does not help in overcoming an inexpressive background theory.

From a proof-theoretic and model-theoretic point of view, the discussion of completeness becomes more interesting. We know that there is a sound and complete, finitary proof system for first-order logic. We can combine that proof system with the proof system for Hoare's logic: any proof needed in the consequence rule can be provided by a deduction in the proof system for first-order logic. The relative completeness result now no longer needs a background theory that is maximally consistent (the oracle), since by the completeness of first-order logic we already know that every semantic consequence (of the background theory) can be deduced. In this case, it is important to keep in mind that program theories are interpreted with a semantics of programs with respect to *arbitrary* structures that satisfies the background theory. This, in fact, further shows that expressivity of the weakest precondition is essential to the completeness result.

Nothing prevents us from taking one of the axiomatic set theories (such as Zermelo-Fraenkel set theory [49], Quine's New Foundations [85], Von Neumann–Bernays–Gödel set theory [153], among others) as a background theory. The resulting program logic is very expressive: we can specify very rich specifications of programs. However, in that case, the relation with practical computing becomes less clear, although even Dijkstra did not mind speaking about programs that work on sets or real numbers.[1] It is an interesting avenue to see what assumptions (such as encodability, recursive enumerability, and decidability) are needed for modeling the primitive operations and tests out of which a program is constructed. One assumption, for example, could be that the value of every accessible or changed program variable must have a digital encoding, so it can actually be represented in the memory of a classical digital computer. However, alternative assumptions may be needed for programs intended to be executed by quantum computers.

Note that we can also prove relative completeness using the strongest postcondition, but we leave that as an exercise for the reader. Also note that we did not need all proof rules in the (relative) completeness proof, but this changes after introducing recursively defined procedures with parameters. For **while**-programs it is in fact the case that the invariance rule, substitution rule, and ∃-introduction rule are admissible. However, these rules are not derivable from the other rules.

[1] `https://www.youtube.com/watch?v=GX3URhx6i2E`

B.5 Recursive procedures

This section sketches how to extend our approach to recursive procedures with parameters. The purpose is to demonstrate that the set-up of the axiomatic semantics above naturally extends to giving a proof system for programs with recursive procedures. We shall limit our formal development to outline the main ideas, and instead refer readers to the journal article *Completeness and Complexity of Reasoning about Call-by-Value in Hoare Logic* for all technical details [29]. However, the presentation given here is slightly more elegant than that of [29], due to the use of program signatures.

The axiomatic semantics above naturally leads to a programming methodology called *design by contract* [154, 18]. In essence, every operation of the program can be assigned a contract that declares a precondition (which the caller of the procedure needs to guarantee) and a postcondition (which the caller of the procedure may assume to hold after the procedure terminates). We follow this methodology in the design of a proof system that supports verifying programs with recursive procedures.

Given a program signature. A recursive program $(D \mid S)$ consists of a *main statement* S and a set of declarations D. A *declaration* declares the meaning of an operation of the program signature, by defining it in terms of a *procedure body* which is a statement of our language. Operations that lack such a declaration are so-called *native* operations. A native operation thus lacks a procedure body. The set D associates to each operation at most one declaration. Declarations are denoted as follows

$$O\langle x_1, \ldots, x_n, \overline{y_1}, \ldots, \overline{y_m}\rangle :: S.$$

The operation is annotated with a set of polarized variables, that indicate that execution of the operation may access variables $x_1, \ldots, x_n$ and may change variables $y_1, \ldots, y_m$. In the context of a set D, we call an operation O for which there is a declaration in D a procedure, and say it has procedure body S. Since the operations are already fixed by the program signature, they can occur in statements, including the main statement.

Intuitively, a procedure body should only access and change the variables as declared. A recursive program is *well-formed* if for all procedures O with body S, we have $\mathit{access}(S) \subseteq \mathit{access}(O)$ and $\mathit{change}(S) \subseteq \mathit{change}(O)$.

Example B.5.1. Given a program signature which has the operations Z, S, P, $+$, and the test $Z?$ that accesses variable x. The following procedure declarations D:

$$\begin{aligned}
Z\langle \overline{z}\rangle &:: \textbf{native}\\
S\langle x, \overline{z}\rangle &:: \textbf{native}\\
P\langle x, \overline{z}\rangle &:: \textbf{native}\\
+\langle x, y, z, w, \overline{x}, \overline{y}, \overline{z}, \overline{w}\rangle &:: \textbf{if } Z?(x) \textbf{ then } z := y \textbf{ else}\\
&\qquad P; w := z; x := y; S; y := z; x := w; +\\
&\ \ \textbf{fi}
\end{aligned}$$

and main statement $+$ forms the recursive program $(D \mid +)$. The intended data structure is that of the natural numbers. The intuition is that Z resets the variable x to the value 0, S computes the successor of the value in x and stores that in z, P computes the predecessor of the value in x and stores it in z (and if x is 0 it does not terminate). We are then able to give the procedure body of $+$ that computes the addition of the values in x and y and stores the result in z. *End of Example.*

For notational convenience, one may leave out the access and change variables in procedure declarations, since the smallest sets of polarized variables can be computed for a given set of declarations. Hence, we shall not write these variable annotations anymore.

Up to now all variables are *global*, in the sense that the same variable in every context refers to the same 'storage location'. We extend the programming language with a *block statement* for introducing *local* variables, which allow us to temporarily change the value of a variable within the scope of the block. As such, the statements are extended to include the complex block statement

$$S ::= \ldots \mid \textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}$$

where $\vec{x}$ and $\vec{y}$ are sequences of variables of the same length, and $\vec{x}$ consists of unique variables. The variables of $\vec{x}$ are local variables within the scope of the block. We also extend the definitions of *access* and *change* as follows:

$$\begin{aligned} \mathit{access}(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}) &= (\mathit{access}(S) \setminus \vec{x}) \cup \vec{y}, \\ \mathit{change}(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}) &= \mathit{change}(S) \setminus \vec{x}. \end{aligned}$$

From the perspective of operational semantics, there is a problem with giving the block statement a small-step semantics. The small-step semantics is in a sense a 'local' semantics, which transforms the statement and state one step at a time. However, the intended semantics is that after the block is exited, the values of the local variables have to be restored to their original value, that is, at the time before entering the block statement. A possible solution is to keep track of the original values in the state by means of a stack, to which values can be pushed, and from which original values can be popped. In that way, the block construct can be seen as a structured short-hand of a program in a block program signature, where the original value of each local variable is first pushed, then the parallel assignment is performed, and after the block ends the original values are popped again in reverse order.

For logical machine models, the big-step semantics of the block statement can be given directly, i.e. without pushing and popping, by the following transition:

$$\begin{aligned} &(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}, s) \twoheadrightarrow (\checkmark, s'[\vec{x} := s(\vec{x})]) \text{ if } (S, s[\vec{x} := \vec{y}]) \twoheadrightarrow (\checkmark, s') \\ &(\textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}, s) \twoheadrightarrow \textbf{fail} \text{ if } (S, s[\vec{x} := \vec{y}]) \twoheadrightarrow \textbf{fail} \end{aligned}$$

where we have parallel update of, and parallel access from, proper states s (being valuations of the underlying structure). These are denoted $s[\vec{x} := \vec{v}]$ for the updated state where $\vec{v}$ is a sequence of new values (of the same length as $\vec{x}$), and $s(\vec{x})$ for a sequence of values that the variables of $\vec{x}$ have in state s, respectively. Without much difficulty it is also possible to extend the denotational semantics in a similar way.

Having local variables and block statements allows us to introduce procedures with parameters. In the signature, for each operation O we also record its arity (being a set of variables), denoted $\mathit{arity}(O)$. Consequently, we extend the notion of declarations to display the arity of operations as follows:

$$O(x_1, \ldots, x_n) :: S,$$

where $x_1, \ldots, x_n$ are distinct variables, exactly covering $\mathit{arity}(O)$, called the *formal parameters* of the operation O. The difference between the arity of an operation, and the formal parameters of an operation is that in the latter the order of variables is significant. (Note that the signature still assigns polarized variables to each operation, indicating the potentially accessible and changed variables, but they are left implicit in declarations.) The formal parameters are local to the procedure body, and thus are never included in the variables that are potentially accessed or changed: the latter variables are global variables that are not among the formal parameters. As such, we require that $\mathit{arity}(O)$ and $\mathit{access}(O)$ are disjoint, as well as $\mathit{arity}(O)$ and $\mathit{change}(O)$.

Further, we have the following call statement:

$$S ::= \ldots \mid O(y_1, \ldots, y_n)$$

where $y_1, \ldots, y_n$ are the *actual parameters* supplied as part of the call, where we assume O has the arity $\vec{x}$, and $\vec{y}$ and $\vec{x}$ of equal length. It is permissible that the actual parameters $\vec{y}$ have duplicate variables, while this is not the case for the formal parameters $\vec{x}$. We speak of a *procedure call* in the context of a set of declarations if the corresponding operation is a procedure, and otherwise speak of a *native call.* We also extend the definitions of *access* and *change* as follows:

$$\begin{aligned} \mathit{access}(O(\vec{y})) &= \mathit{access}(O) \cup \vec{y}, \\ \mathit{change}(O(\vec{y})) &= \mathit{change}(O). \end{aligned}$$

Intuitively, a procedure body should only access and change the variables as declared, but it is permissible to access and change the formal parameters. A recursive program is *well-formed* if for all procedures O with body S, we have $(\mathit{access}(S) \setminus \mathit{arity}(O)) \subseteq \mathit{access}(O)$ and $(\mathit{change}(S) \setminus \mathit{arity}(O)) \subseteq \mathit{change}(O)$.

In fact, procedures without parameters can be regarded as procedures with zero parameters, that have an empty arity. The notions defined *with* parameters thus are a refinement of the former notions *without* parameters.

To give semantics to recursive programs, we lift the semantics of statements to a semantics of (well-formed) recursive programs by also considering the set of declarations in each configuration. The big-step semantics of the procedure call then is defined by the transition

$$(D \mid O(\vec{y}), s) \longrightarrow\!\!\!\!\!\rightarrow C \text{ if } (D \mid \textbf{begin local } \vec{x} := \vec{y}; S \textbf{ end}, s) \longrightarrow\!\!\!\!\!\rightarrow C$$

where $O(\vec{x}) :: S$ is in the set of declarations D. We have that $\vec{x}$ and $\vec{y}$ match in length, because the actual parameters and arity of an operation match in length and also that the formal parameters gives an order of the variables of the arity. This style of operational semantics is also called *body replacement*, since intuitively we replace the procedure call by the body of the procedure wrapped in a block statement. We here restrict ourselves to the call-by-value parameter passing mechanism, meaning that the values of actual parameters are computed before starting to execute the procedure body. This is in contrast to call-by-name where we would have to perform substitution of formal parameters by actual parameters in the procedure body, resulting in a different replacement for each procedure call with different actual parameters, and requires handling variable capturing by block statements. Note that the behavior of a native call is left unspecified, and as such can be interpreted by the operationalization of a machine model.

The denotational semantics can be given iteratively. This is similar to how we previously gave a denotational semantics to **while**-statements. We leave the details out and instead refer the reader to [29]. What is more important concerning our discussion is the proof system for recursive programs.

Procedures return values by the use of a global variables (that can be changed). It is possible to designate a special variable, called **result**, that cannot occur as a formal parameter of any procedure declaration, and to which the procedure may assign an output value in the procedure body. For procedure declarations that have the **result** variable listed among its changed variables, we can introduce the following abbreviation:

$$x := O(\vec{y})$$

which denotes the statement

$$O(\vec{y}); x := \mathbf{result}.$$

Since we have introduced blocks that allow us to introduce local variables, we can also introduce the following syntactic sugar. An *expression* e is constructed out of either an individual variable, or a procedure call with as formal parameters other expressions (matching in length the number of formal parameters of the corresponding operation). For example, $O(P(x, y), z, Q(w))$ is an expression given operations O, P, Q and variables x, y, z, w. Each expression $O(e_1, \ldots, e_n)$ abbreviates a statement. If all expressions $e_1, \ldots, e_n$ are variables then $O(e_1, \ldots, e_n)$ is simply a procedure call with the corresponding variables as actual parameters. Otherwise, let e_i be the first non-variable expression from the left, then $O(e_1, \ldots, e_n)$ abbreviates the following statement:

$$e_i; \mathbf{begin\ local}\ z_i := \mathbf{result}; O(e_1, \ldots, z_i, \ldots, e_n)\ \mathbf{end}$$

and the statement $x := O(e_1, \ldots, e_n)$ abbreviates

$$O(e_1, \ldots, e_n); x := \mathbf{result}.$$

where we take $z_1, \ldots, z_n$ to be fresh variables (i.e. not occurring in any of the sub-expressions, nor in accessible or changed variables of the operation). Note that this effectively evaluates the expressions from the left to the right, storing the result of each sub-expression in a local variable. Also note that the abbreviation is recursively defined.

Note the resemblance between terms (in the assertion language) and expressions (in the programming language). Terms can be added to a logic without terms, by the introduction of an existential quantifier that binds the output value that are assigned to inputs by a functional relation. In that way, terms can be used in the place of variables. Similarly, we use local variables to capture the output of an expression so that, as well, expressions can be used in the place of variables. However, note that with expressions we also defined a so-called *order of evaluation*, by evaluating the expressions from left to right. The reason to do so, and why this is not needed in the assertion language, is because expressions can have *side-effects* on global variables, which consequently affect the evaluation of subsequent expressions.

Also note that the programming language as described above employs *dynamic scoping*. This means that a global variable can be captured by the use of a block statement, to isolate the effects of procedures. We say that a program is *statically scoped* if the local variables and global variables are separated, thus disallowing capturing of global variables and ensuring that the referent of a global variable remains the same in every context. This may remind the reader of Barendregt's variable convention, in which bound and free variables are separate. See also the discussion of local variables by K.R. Apt and E.-R. Olderog [11, Section 5.2].

Finally, we consider extending the proof system **HL** with rules for proving properties about block statements and procedure calls. The objects of the proof system remain Hoare triples $\{\phi\}\ S\ \{\psi\}$ in which we are oblivious of the set of declarations in the proof system. We furthermore add the Hoare triples $\{\phi\}\ D \mid S\ \{\psi\}$ for specifying (well-formed) recursive programs $(D \mid S)$.

We have the following additional proof rules:

(block) $\{\phi[\vec{x} := \vec{y}]\}\ S\ \{\psi\} \vdash^{\mathbf{HL}} \{\phi\}\ \textbf{begin local}\ \vec{x} := \vec{y}; S\ \textbf{end}\ \{\psi\}$ if $FV(\psi) \cap \vec{x} = \emptyset$,

(inst) $\{\phi\}\ O(\vec{x})\ \{\psi\} \vdash^{\mathbf{HL}} \{\phi[\vec{x} := \vec{y}]\}\ O(\vec{y})\ \{\psi\}$ if $FV(\psi) \cap \vec{x} = \emptyset$,

(rec) If $\Gamma, \Delta \vdash^{\mathbf{HL}} \{\phi_i\}\ S_i\ \{\psi_i\}$ for every $1 \leq i \leq n$ and $\Gamma, \Delta \vdash^{\mathbf{HL}} \{\phi\}\ S\ \{\psi\}$, then
$\Gamma \vdash^{\mathbf{HL}} \{\phi\}\ D \mid S\ \{\psi\}$ where $D = \{O_1(\vec{x}_1) :: S, \ldots, O_n(\vec{x}_n) :: S_n\}$
and $\Delta = \{\{\phi_1\}\ O_1(\vec{x}_1)\ \{\psi_1\}, \ldots, \{\phi_n\}\ O_n(\vec{x}_n)\ \{\psi_n\}\}$
and $FV(\psi_i) \cap \vec{x}_i = \emptyset$ for every $1 \leq i \leq n$.

Recall that $\phi[\vec{x} := \vec{y}]$ is the substitution of the variables $\vec{x}$ by the corresponding variables $\vec{y}$. The specifications in Δ are called *contracts*, and we require that the formal parameters of each procedure do not occur in the postcondition of the contract. The difference between Γ and Δ is that Γ consists of the program theory and background theory, used for axiomatizing the native operations and the underlying structure, whereas Δ introduces contracts for the procedures which have a procedure body.

It is now possible to formulate both soundness and (relative) completeness for recursive programs too. The essence of the relative completeness proof is to introduce *most general contracts* for each procedure, making use of a strongest postcondition axiomatization in the line of Gorelick [96]. We refer the reader to [29] for more details.

Remark B.5.2. These rules are admissible in Hoare's logic: every local block can be eliminated by introducing fresh variables (that do not occur in any surrounding program), and recursive procedures can be eliminated and reduced to simple **while** loops. However, showing why this is the case in detail is out of scope of this thesis.

Appendix C

Intuitionistic separation logic

C.1 Standard semantics

Definition C.1.1 (Satisfaction relation)**.** f Given a structure $\mathfrak{A} = (A, \mathcal{I})$, a valuation ρ of $\mathfrak{A}$, a finite heap h of $\mathfrak{A}$, and a separation logic formula ϕ. The satisfaction relation $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \phi$ is defined inductively on the structure of ϕ:

- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \bot$ never holds,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} (x \doteq y)$ iff $\rho(x) = \rho(y)$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} (x \hookrightarrow y)$ iff $h(\rho(x))$ is defined and $h(\rho(x)) = \rho(y)$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} C(x_1, \ldots, x_n)$ iff $(\rho(x_1), \ldots, \rho(x_n)) \in C^{\mathcal{I}}$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \phi \to \psi$ iff $\mathfrak{A}, h', \rho \models^{\mathbf{SISL}} \phi$ implies $\mathfrak{A}, h', \rho \models^{\mathbf{SISL}} \psi$ for every $h' \supseteq h$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \forall x \phi$ iff $\mathfrak{A}, h, \rho[x := a] \models^{\mathbf{SISL}} \phi$ for every $a \in A$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \phi * \psi$ iff $\mathfrak{A}, h_1, \rho \models^{\mathbf{SISL}} \phi$ and $\mathfrak{A}, h_2, \rho \models^{\mathbf{SISL}} \psi$ for some h_1, h_2 such that $h \equiv h_1 \uplus h_2$,
- $\mathfrak{A}, h, \rho \models^{\mathbf{SISL}} \phi \mathrel{-\!\!*} \psi$ iff $\mathfrak{A}, h', \rho \models^{\mathbf{SSL}} \phi$ implies $\mathfrak{A}, h'', \rho \models^{\mathbf{SISL}} \psi$ for every h', h'' such that $h'' \equiv h \uplus h'$.

This definition is based on finite heaps. **SISL** stands for Standard Intuitionistic Separation Logic. It crucially differs from **SSL** in the clause for logical implication. Note that the pure fragment of separation logic still is interpreted classically, since pure formulas do not depend on the heap. The definition above can be adapted to obtain full intuitionistic separation logic, and general intuitionistic separation logic, in a similar manner as before.

C.2 Intuitionistic Reynolds' logic

In the intuitionistic version of separation logic we cannot express directly anymore that a location x is not allocated. The definition of the substitution $p[\langle x\rangle := e]$ and $p[\langle x\rangle := \bot]$, and the above *alt-backwards* axiomatization of mutation, allocation and dispose instructions therefore breaks down. We can use new modalities $[[x] := e]$ and $[x] := \bot]$ corresponding to the mutation and the dispose instruction. Differently from the heap update operation and the heap clear operation, correctness of the modalities $[[x] := e]$ and $[[x] := \bot]$ require that x *is* allocated. In ISL we then can define the heap update substitution $p[\langle x\rangle := e]$ by $(x \hookrightarrow -) \to p[[x] := e]$, as explained in details below.

Further note, that we indeed can use in the allocation axiom disjunction (instead of the intuitionistic implication) because of its classic interpretation (this is explained in the soundness and completeness proof below).

Definition C.2.1 (Substitution for mutation)**.** We define $p[[x] := e]$ recursively on p (assuming the variables of e and x do not occur bound in p).

- $b[[x] := e] = b$,
- $(e' \hookrightarrow e'')[[x] := e] = (x \neq e' \wedge e' \hookrightarrow e'') \vee (x = e' \wedge e'' = e)$,
- $(p \wedge q)[[x] := e] = p[[x] := e] \wedge q[[x] := e]$, and similar for $\vee$ and $\to$,
- $(\exists y p)[[x] := e] = \exists y(p[[x] := e])$ and similar for $\forall$,
- $(p * q)[[x] := e] = ((p[[x] := e] \wedge x \hookrightarrow -) * q) \vee (p * (q[[x] := e] \wedge x \hookrightarrow -))$
- $(p \mathrel{-\!\!*} q)[[x] := e] = p \mathrel{-\!\!*} (q[[x] := e])$

Lemma C.2.2 (Correctness mutation substitution)**.** *Let $s(x) \in dom(h)$. We then have $h, s \models p[[x] := e]$ iff $h[s(x) := s(e)], s \models p$.*

Proof. The proof proceeds by induction on the structure of p. We treat the following main cases.

- $h, s \models (p \to q)[[x] := e]$ iff (definition substitution)
  $h, s \models p[[x] := e] \to q[[x] := e]$ iff (semantics implication)
  $h', s \models p[[x] := e]$ implies $h', s \models q[[x] := e]$, for all h' such that $h \sqsubseteq h'$
  iff (induction hypothesis)
  $h'[s(x) := s(e)], s \models p$ implies $h'[s(x) := s(e)], s \models q$, for all h' such that $h \sqsubseteq h'$
  iff (see below)
  $h', s \models p$ implies $h', s \models q$, for all h' such that $h[s(x) := s(e)] \sqsubseteq h'$
  iff (semantics implication)
  $h[s(x) := s(e)], s \models p \to q$
  Note that $h[s(x) := s(e)] \sqsubseteq h'$ implies $h'[s(x) := h(s(x))] = h'$, and $h \sqsubseteq h'$ implies $h[s(x) := s(e)] \sqsubseteq h'[s(x) := s(e)]$.

- $h, s \models (p * q)[[x] := e]$
  iff (definition substitution)
  $h, s \models (x \hookrightarrow -) \rightarrow ((p[[x] := e] \wedge x \hookrightarrow -) * q) \vee (p * (q[[x] := e] \wedge x \hookrightarrow -))$
  iff (see below)
  $h[s(x) := s(e)], s \models p * q$.
  $\Downarrow$: First, let $s(x) \in dom(h)$. W.l.o.g. we may assume that $h = h_1 \uplus h_2$, $h_1, s \models p[[x] := e]$, and $h_2, s \models q$, for some h_1, h_2 such that $s(x) \in dom(h_1)$. Induction hypothesis: $h_1[s(x) := s(e)], s \models p$. Further: $h[s(x) := s(e)] = h_1[s(x) := s(e)] \uplus h_2$. Next, let $s(x) \notin dom(h)$. Let $h' = h[s(x) := n]$, for some arbitrary n. Again, w.l.o.g. we may assume that $h' = h_1 \uplus h_2$, $h_1, s \models p[[x] := e]$, and $h_2, s \models q$, for some h_1, h_2 such that $s(x) \in dom(h_1)$. Induction hypothesis: $h_1[s(x) := s(e)], s \models p$. Further: $h'[s(x) := s(e)] = h[s(x) := s(e)] = h_1[s(x) := s(e)] \uplus h_2$.
  $\Uparrow$: Let $h[s(x) := s(e)] = h_1 \uplus h_2$ such that $h_1, s \models p$ and $h_2, s \models q$. W.l.o.g., assume that $s(x) \in dom(h_1)$. Further , let $h \sqsubseteq h'$. Let $h'_1 = h' \setminus h_2$. It follows that $h' = h'_1 \uplus h_2$. Monotonicity: $h'_1, s \models p$. Induction hypothesis: $h'_1, s \models p[[x] := e]$ (note that $h'_1[s(x) := s(e)] = h'_1$).

- $h, s \models (p -\!\!* q)[[x] := e]$
  iff (definition of substitution)
  $h, s \models (x \hookrightarrow -) \rightarrow (p -\!\!* (q[[x] := e]))$ iff (semantics intuitionistic and separating implication)
  $h', s \models p$ implies $h \uplus h', s \models q[[x] := e]$, for every $h \subseteq h'$, with $s(x) \in dom(h')$, and h'' disjoint from h'
  iff (induction hypothesis)
  $h'', s \models p$ implies $(h' \uplus h'')[s(x) := s(e)], s \models q$, for every $h \subseteq h'$, with $s(x) \in dom(h')$, and h'' disjoint from h'
  iff (see below)
  $h', s \models p$ implies $h[s(x) := s(e)] \uplus h', s \models q$, for every h' disjoint from $h[s(x) := s(e)]$
  iff (semantics of separating implication)
  $h[s(x) := s(e)], s \models p -\!\!* q$.
  $\Downarrow$: First, let $s(x) \notin dom(h)$. So $h \sqsubseteq h[s(x) := s(e)]$, and we can take $h[s(x) := s(e)]$ for h'. Next, let $s(x) \in dom(h)$. So it suffices to observe that $h''\#h$ implies $h''\#h[s(x) := s(e)]$.
  $\Uparrow$: let $h \subseteq h'$, with $s(x) \in dom(h')$, and h'' disjoint from h' such that $h'', s \models p$. Clearly, h'' is disjoint from $h[s(x) := s(e)]$, and thus we have that $h[s(x) := s(e)] \uplus h'', s \models q$, that is, $(h \uplus h'')[s(x) := s(e)], s \models q$. We further have that $(h \uplus h'')[s(x) := s(e)] \sqsubseteq (h; \uplus h'')[s(x) := s(e)]$, and so by monotonicity, we conclude that $(h' \uplus h'')[s(x) := s(e)], s \models q$. □

Corollary C.2.3 (Correctness intuitionistic heap update)**.**
We have $h, s \models p[\langle x \rangle := e]$ *iff* $h[s(x) := s(e)], s \models p$.

Proof. First let $h, s \models p[\langle x \rangle := e]$, that is (by definition), $h, s \models (x \hookrightarrow -) \rightarrow p[[x] := e]$. Let $h' = h$, if $s(x) \in dom(h)$, and $h' = h[s(x) := n]$, for some arbitrary n, otherwise. So $h \sqsubseteq h'$, and thus we infer from $h, s \models (x \hookrightarrow -) \rightarrow p[[x] := e]$ that

$h', s \models p[[x] := e]$, and so by the correctness of the mutation substitution, we have $h'[s(x) := s(e)], s \models p$, that is, $h[s(x) := s(e)], s \models p$.

On the other hand, assuming $h[s(x) := s(e)], s \models p$, let $h \subseteq h'$ such that $s(x) \in dom(h')$. We show that $h', s \models p[[x] := e]$: By the monotonicity property of ISL we have that $h[s(x) := s(e)], s \models p$ implies $h'[s(x) := s(e)], s \models p$. By the correctness of the mutation substitution, it then suffices to observe that $h', s \models p[[x] := e]$ if and only if $h'[s(x) := s(e)], s \models p$. □

For the intutitionistic axiomatization of the dispose instruction we introduce the following substitution.

Definition C.2.4 (Substitution for dispose)**.** We define $p[[x] :=\perp]$ recursively on p (assuming that x does not occur bound in p).

- $b[[x] :=\perp] = b$
- $(e \hookrightarrow e')[[x] :=\perp] = x \neq e \wedge e \hookrightarrow e'$
- $(p \wedge q)[[x] :=\perp] = p[[x] :=\perp] \wedge q[[x] :=\perp]$, and similar for $\vee$
- $(p \rightarrow q)[[x] :=\perp] = (p[[x] :=\perp] \rightarrow q[[x] :=\perp]) \wedge \forall y(p[[x] := y] \rightarrow q[[x] := y])$ where y is a fresh variable
- $(\exists y p)[[x] :=\perp] = \exists y(p[[x] :=\perp])$
- $(p * q)[[x] :=\perp] = (p[[x] :=\perp] \wedge (x \hookrightarrow -)) * q$
- $(p \mathrel{-\!\!*} q)[[x] :=\perp] = (p \mathrel{-\!\!*} q[[x] :=\perp]) \wedge \forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$ where y is a fresh variable.

Determining whether $p * q$ holds after disposing x, we predict whether p or q holds for the sub-heap that contained the disposed x. Since the dispose instruction $[x] :=\perp$ requires that x is allocated, we distinguish between between these two cases by checking in which part of the heap x is allocated. But since after the dispose instruction both p and q are evaluated in sub-heaps which do not contain the location x, we can choose between where to allocate x initially. Formally, the assertions $(p[[x] :=\perp] \wedge (x \hookrightarrow -)) * q$ and $(q[[x] :=\perp] \wedge (x \hookrightarrow -)) * p$ are equivalent. For example, we that $\mathbf{true} * (x \hookrightarrow y)$ does not hold after execution of $[x] :=\perp$. By definition $(\mathbf{true} * (x \hookrightarrow y))[[x] :=\perp]$ reduces to $(\mathbf{true} \wedge x \hookrightarrow y) * (x \hookrightarrow y)$, which further reduces to **false**. On the other hand, $((x \hookrightarrow y) * \mathbf{true})[[x] :=\perp]$ reduces to $(x \neq x \wedge (x \hookrightarrow -)) * \mathbf{true}$, which further reduces to $\mathbf{false} * \mathbf{true}$, which also is equivalent to **false**.

Lemma C.2.5 (Correctness dispose substitution)**.** *Let $s(x) \in dom(h)$. We then have $h, s \models p[[x] :=\perp]$ if and only if $h[s(x) := \perp], s \models p$.*

Proof. The proof proceeds by induction on the structure of p. We treat the following main cases.

- $h, s \models (p \to q)[[x] :=\perp]$ iff (definition of substitution)
  $h, s \models (p[[x] :=\perp] \to q[[x] :=\perp]) \wedge \forall y(p[[x] := y] \to q[[x] := y])$
  iff (see below)
  $h[s(x) :=\perp], s \models p \to q$.
  First we show that $h[s(x) :=\perp] \sqsubseteq h'$ and $h', s \models p$ implies $h', s \models q$. We distinguish the following two cases. First let $s(x) \notin dom(h')$. It follows that $h'[s(x) := h(s(x))], s \models p[[x] :=\perp]$ (by the induction hypothesis we have $h'[s(x) := h(s(x))], s \models p[[x] :=\perp]$ if and only if $h', s \models p$). Since $h \sqsubseteq h'[s(x) := h(s(x))]$ we thus derive from $h, s \models (p[[x] :=\perp] \to q[[x] :=\perp])$ that $h'[s(x) := h(s(x))], s \models q[[x] :=\perp]$, and so by the induction hypothesis again we obtain $h', s \models q$.

  Next let $s(x) \in dom(h')$. From $h', s \models p$, it follows from the correctness of the mutation substitution that $h'[s(x) := h(s(x))], s' \models p[[x] := y]$, where $s' = s[y := h'(s(x))]$ (since y does not appear in p and $x \neq y$). Since $h \sqsubseteq h'[s(x) := h(s(x))]$ we thus derive from $h, s' \models p[[x] := y] \to q[[x] := y]$ that $h'[s(x) := h(s(x))], s' \models q[[x] := y]$, and so by the correctness of the mutation substitution again we obtain $h', s' \models q$, that is, $h', s \models q$ (since y does not appear in q).

  Conversely, let $h[s(x) :=\perp], s \models p \to q$. First we show that $h, s \models p[[x] :=\perp] \to q[[x] :=\perp]$. Let $h \sqsubseteq h'$ and $h', s \models p[[x] :=\perp]$, and so by the induction hypothesis $h'[s(x) :=\perp], s \models p$. We have to show that $h', s \models q[[x] :=\perp]$. By the induction hypothesis ($s(x) \in dom(h) \subseteq dom(h')$) it suffices to show that $h'[s(x) :=\perp], s \models q$. Since $h[s(x) :=\perp] \sqsubseteq h'[s(x) :=\perp]$ and $h'[s(x) :=\perp], s \models p$ we thus derive from $h[s(x) :=\perp], s \models p \to q$ that $h'[s(x) :=\perp], s \models q$.

  Next we show that $h, s \models \forall y(p[[x] := y] \to q[[x] := y])$. Let $h \sqsubseteq h'$ and $h', s' \models p[[x] := y]$, where $s' = s[y := n]$, for some arbitrary n. We have to show that $h', s' \models q[[x] := y]$ which by the correctness of the mutation substitution boils down to $h'[s(x) := n], s \models q$. By the correctness of the mutation substitution again we have $h', s' \models p[[x] := y]$ if and only if $h'[s(x) := n], s' \models p$. Since $h[s(x) :=\perp] \sqsubseteq h'[s(x) := n]$ we thus derive from $h[s(x) :=\perp], s \models p \to q$ and $h'[s(x) := n], s' \models p$ that $h'[s(x) := n], s \models q$ (since y does not occur in p and q).

- $h, s \models (p * q)[[x] :=\perp]$ iff (definition substitution)
  $h, s \models (p[[x] :=\perp] \wedge x \hookrightarrow -) * q$ iff (semantics separating conjunction)
  $h_1, s \models p[[x] :=\perp] \wedge x \hookrightarrow -$ and $h_2, s \models q$, for some h_1, h_2 such that $h = h_1 \uplus h_2$
  iff (semantics of points-to) $h_1, s \models p[[x] :=\perp]$ and $h_2, s \models q$, for some h_1, h_2 such that $s(x) \in dom(h_1)$ and $h = h_1 \uplus h_2$
  iff (induction hypothesis)
  $h_1[s(x) :=\perp], s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $s(x) \in dom(h_1)$ and $h = h_1 \uplus h_2$
  iff (see below)
  $h_1, s \models p$ and $h_2, s \models q$, for some h_1, h_2 such that $h = h_1 \uplus h_2$
  iff (semantics separating conjunction)
  $h[s(x) :=\perp], s \models p * q$. Note that $s(x) \in dom(h_1)$ and $h = h_1 \uplus h_2$ implies

$h[[x] :=\perp] = h_1[s(x) :=\perp] \uplus h_2$, and $h[[x] :=\perp] = h_1 \uplus h_2$ implies that $h = h_1[s(x) := h(s(x))] \uplus h_2$.

- $h, s \models ((p \mathrel{-\!\!*} q)[[x] :=\perp]$
iff (definition substitution)
$h, s \models (p \mathrel{-\!\!*} q[[x] :=\perp]) \wedge \forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$
iff (see below)
$h[s(x) :=\perp], s \models p \mathrel{-\!\!*} q$.
First let h' be disjoint from $h[s(x) :=\perp]$ and $h', s \models p$. We have to show that $h[s(x) :=\perp] \uplus h', s \models q$. We distinguish the following two cases.

First, let $s(x) \notin dom(h')$. So h' and h are disjoint, and thus (since $h, s \models p \mathrel{-\!\!*} q[[x] :=\perp]$) we have $h \uplus h', s \models q[[x] :=\perp]$. From which we derive $(h \uplus h')[s(x) :=\perp], s \models q$ by the induction hypothesis (note that $s(x) \in dom(h) \subseteq dom(h \uplus h')$). We then can conclude this case by the observation that $h[s(x) :=\perp] \uplus h' = (h \uplus h')[s(x) :=\perp]$.

Next, let $s(x) \in dom(h')$. We then introduce $s' = s[y := h'(s(x))]$. Since $h', s' \models p$ (y does not appear in p), it follows by the correctness of the intuitionistic heap update (see above corollary) that $h'[s(x) :=\perp], s' \models p[\langle x\rangle := y]$. Since $h'[s(x) :=\perp]$ and h are disjoint (which clearly follows from that h' and $h[s(x) :=\perp]$ are disjoint), and so (since $h, s' \models p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y]$) we have that $h \uplus (h'[s(x) :=\perp]), s' \models q[\langle x\rangle := y]$. Applying again the correctness of the intuitionistic heap update we obtain $(h \uplus (h'[s(x) :=\perp]))[s(x) := s'(y)], s' \models q$. We then can conclude this case by the assumption that y does not appear in q and the observation that $h[s(x) :=\perp] \uplus h' = (h \uplus (h'[s(x) :=\perp]))[s(x) := s'(y)]$.

Conversely, let $h[s(x) :=\perp], s \models p \mathrel{-\!\!*} q$. We first show that $h, s \models p \mathrel{-\!\!*} q[[x] :=\perp]$: Let h' be disjoint from h and $h', s \models p$. We have to show that $h \uplus h', s \models q[[x] :=\perp]$. Clearly, h' and $h[s(x) :=\perp]$ are disjoint, and so (since $h[s(x) :=\perp], s \models p \mathrel{-\!\!*} q$) $h[s(x) :=\perp] \uplus h', s \models q$. By the induction hypothesis (note that $s(x) \in dom(h) \subseteq dom(h \uplus h')$) we have $h \uplus h', s \models q[[x] :=\perp]$ iff $(h \uplus h')[s(x) :=\perp], s \models q$. We then can conclude this case by the observation that $(h \uplus h')[s(x) :=\perp] = h[s(x) :=\perp] \uplus h'$, because $s(x) \in dom(h) \setminus dom(h')$.

Next we show that $h, s \models \forall y(p[\langle x\rangle := y] \mathrel{-\!\!*} q[\langle x\rangle := y])$: Let h' be disjoint from h and $s' = s[y := n]$, for some n, such that $h', s' \models p[\langle x\rangle := y]$. We have to show that $h \uplus h', s' \models q[\langle x\rangle := y]$. By the correctness of the intuitionistic heap update it then follows that $h'[s(x) := n], s' \models p$, that is, $h'[s(x) := n], s \models p$ (since y does not appear in p). Since $h'[s(x) := n]$ and $h[s(x) :=\perp]$ are disjoint, we derive from the assumption $h[s(x) :=\perp], s \models p \mathrel{-\!\!*} q$ that $h[s(x) :=\perp] \uplus h'[s(x) := n], s \models q$. Again by the correctness of the intuitionistic heap update, we have that $h \uplus h', s' \models q[\langle x\rangle := y]$ iff $(h \uplus h')[s(x) := n], s' \models q$ (that is, $(h \uplus h')[s(x) := n], s \models q$, because y does not appear in q). We then can conclude this case by the observation that $(h \uplus h')[s(x) := n] = h[s(x) :=\perp] \uplus h'[s(x) := n]$. □

The following axiomatization is by Reynolds [187].

Definition C.2.6 (Weakest precondition axiomatization).

$$\{p[x := e]\}\ x := e\ \{p\}$$

$$\{\exists y((e \hookrightarrow y) \wedge p[x := y])\}\ x := [e]\ \{p\}$$

$$\{(x \hookrightarrow -) * ((x \hookrightarrow e) \mathrel{-\!\!*} p)\}\ [x] := e\ \{p\}$$

$$\{\forall y((y \hookrightarrow e) \mathrel{-\!\!*} p[x := y])\}\ x := \mathbf{new}(e)\ \{p\}$$

$$\{(x \hookrightarrow -) * p\}\ \mathbf{delete}(x)\ \{p\}$$

The next two axiomatizations are novel.

Definition C.2.7 (Alternative weakest precondition axiomatization).

$$\{p[x := e]\}\ x := e\ \{p\}$$

$$\{\exists y((e \hookrightarrow y) \wedge p[x := y])\}\ x := [e]\ \{p\}$$

where y is fresh

$$\{(x \hookrightarrow -) \wedge p[[x] := e]\}\ [x] := e\ \{p\}$$

$$\{\forall x((x \hookrightarrow -) \vee p[\langle x \rangle := e])\}\ x := \mathbf{new}(e)\ \{p\}$$

where $x \notin fv(e)$

$$\{(x \hookrightarrow -) \wedge p[[x] := \bot]\}\ \mathbf{delete}(x)\ \{p\}$$

Definition C.2.8 (Strongest postcondition axiomatization).

$$\{p\}\ x := e\ \{(\exists y)(p[x := y] \wedge e[x := y] = x)\}$$

$$\{(e \hookrightarrow -) \wedge p\}\ x := [e]\ \{\exists y(p[x := y] \wedge e[x := y] \hookrightarrow x)\}$$

$$\{(x \hookrightarrow -) \wedge p\}\ [x] := e\ \{\exists y(p[[x] := y]) \wedge (x \hookrightarrow e)\}$$

where $x \notin fv(e)$

$$\{p\}\ x := \mathbf{new}(e)\ \{(\exists y(p[x := y]))[[x] := \bot] \wedge (x \hookrightarrow e)\}$$

$$\{(x \hookrightarrow -) \wedge p\}\ \mathbf{delete}(x)\ \{(x \hookrightarrow -) \to \exists y(p[[x] := y])\}$$

where y is fresh everywhere

We showcase the soundness and completeness of the strongest postcondition axiomatization of dispose (soundness and completeness of the above axiomatization of the other instructions follow in a straightforward manner from the corresponding substitution lemmas).

- $\models \{p \wedge (x \hookrightarrow -)\}$ **delete**(x) $\{(x \hookrightarrow -) \to \exists y(p[[x] := y])\}$:
  Let $h, s \models r \wedge x \hookrightarrow -$. We have to show that $h[s(x) :=\perp], s \models x \hookrightarrow - \to \exists y(p[[x] := y])$. That is, for $h[s(x) :=\perp] \sqsubseteq h'$ such that $s(x) \in \mathit{dom}(h')$ we have to show that $h', s \models \exists y(p[[x] := y])$: We show $h'[s(x) := n], s[y := n] \models p$ for $n = h(s(x))$: By Lemma C.2.2, we have $h', s[y := n] \models p[[x] := y]$ if and only if $h'[s(x) := n], s[y := n] \models p$. By *monotonicity* ($h, s \models p$ and $h \sqsubseteq h'[s(x) := n]$) it follows that $h'[s(x) := n], s \models p$. Since y does not appear in p, we thus have that $h'[s(x) := n], s[y := n] \models p$.

- $\models \{p \wedge (x \hookrightarrow -)\}$ **delete**(x) $\{q\}$ implies $\models ((x \hookrightarrow -) \to \exists y(p[[x] := y])) \to q$:
  This boils does to showing that $h, s \models x \hookrightarrow - \to \exists y(p[[x] := y])$ implies $h, s \models q$, for any heap h and store s. So, let $h, s \models x \hookrightarrow - \to \exists y(p[[x] := y])$, that is, $h', s \models x \hookrightarrow -$ implies $h', s \models \exists y(p[[x] := y])$, for any $h \sqsubseteq h'$. Let $h' = h$, in case $s(x) \in \mathit{dom}(h)$, and $h' = h[s(x) := n]$, for some arbitrary n, otherwise. Clearly, $h \sqsubseteq h'$ and $h', s \models x \hookrightarrow -$. So $h', s \models \exists y(p[[x] := y])$. Let $h', s[y := k] \models p[[x] := y]$, for some k. By Lemma C.2.2 again, it follows that $h'[s(x) := k], s[y := k] \models p$. From our assumption $\models \{p \wedge x \hookrightarrow -\}$ $[x] :=\perp$ $\{q\}$ we then derive that $h'[s(x) :=\perp], s[y := k] \models q$. By definition of h' we have that $h'[s(x) :=\perp] \sqsubseteq h$, and so by monotonicity we infer $h, s[y := k] \models q$, and so $h, s \models q$, assuming w.l.o.g. that y does not appear (free) in q.

Appendix D

Formalization in Coq

The main motivation behind formalizing results in a proof assistant is to rigorously check hand-written proofs. For our formalization we used the dependently-typed calculus of inductive constructions as implemented by the Coq proof assistant.

This thesis is accompanied by an artifact [112]. In this appendix, we discuss two parts of the artifact: one corresponding to the alternative axiomatization of Reynolds' logic (see Section 4.5 and Appendix C) based on the insights from dynamic separation logic (see Section 4.4), and one corresponding to the natural deduction proof system for separation logic (see Section 3.3).

D.1 Alternative axiomatization

In this part of the artifact, we have used no axioms other than the axiom of function extensionality (for every two functions f, g we have that $f = g$ if $f(x) = g(x)$ for all x) and propositional extensionality (equivalent propositions are equal). This means that we work with an underlying intuitionistic logic: we have not used the axiom of excluded middle for reasoning classically about propositions. However, the decidable propositions (propositions P for which the excluded middle $P \vee \neg P$ can be proven) allow for a limited form of classical reasoning.

We formalize the basic instructions of our programming language (assignment, look-up, mutation, allocation, and deallocation) and the semantics of basic instructions. For Boolean and arithmetic expressions we use a shallow embedding, so that those expressions can be directly given as a Coq term of the appropriate type (with a coincidence condition assumed, i.e. that values of expressions depend only on finitely many variables of the store).

There are two approaches in formalizing the semantics of assertions: shallow and deep embedding. We have taken both approaches. In the first approach, the shallow embedding of assertions, we define assertions of DSL by their extension of satisfiability (i.e. the set of heap and store pairs in which the assertion is satisfied), that must satisfy a coincidence condition (assertions depend only on finitely many variables of the store) and a stability condition (see below). The definition of the modality operator follows from the semantics of programs, which includes basic control structures such as the **while**-loop. In the second approach, the deep embedding of assertions, assertions are modeled using an inductive type and we explicitly introduce two meta-operations on assertions that capture the heap update and heap clear modality. We have omitted the clauses for **emp** and $(e \mapsto e')$, since these could be defined as abbreviations, and we restrict to the basic instructions.

In the deep embedding we have no constructor corresponding to the program modality $[S]p$. Instead, two meta-operations denoted $p[\langle x\rangle = e]$ and $p[\langle x\rangle := \bot]$ are defined recursively on the structure of p. Crucially, we formalized and proven the following lemmas (the details are almost the same as showing the equivalences hold in the shallow embedding, Lemmas 4.4.3 and 4.4.4):

Lemma D.1.1 (Heap update substitution lemma)**.**
$h, s \models p[\langle x\rangle := e]$ iff $h[s(x) := s(e)], s \models p$.

Lemma D.1.2 (Heap clear substitution lemma)**.**
$h, s \models p[\langle x\rangle := \bot]$ iff $h[s(x) := \bot], s \models p$.

By also formalizing a deep embedding, we show that the modality operator can be defined entirely on the meta-level by introducing meta-operations on formulas that are recursively defined by the structure of assertions: this captures Theorem 4.4.5. For technical simplicity we restrict ourselves to the basic instructions, but it should be natural to extend the formalization of the completeness result to languages with **while**-statements, e.g. following [81]. On the other hand, in the shallow embedding it is easier to show that our approach can be readily extended to complex programs including **while**-loops.

In both approaches, the semantics of assertions is classical, although we work in an intuitionistic meta-logic. We do this by employing a double negation translation, following the set-up by R. O'Connor [162]. In particular, we have that our satisfaction relation $h, s \models p$ is stable, i.e. $\neg\neg(h, s \models p)$ implies $h, s \models p$. This allows us to do classical reasoning on the image of the higher-order semantics of our assertions.

The source code of our formalization is accompanied with this thesis as a digital artifact. The artifact consists of the following files:

- `shallow/Language.v`: Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a shallow embedding of our assertion language, as presented in the prequel.

- `shallow/Proof.v`: Provides proof of the equivalences (**E1-16**), and additionally standard equivalences for modalities involving complex programs.

- `deep/Heap.v`: Provides an axiomatization of heaps as partial functions.

- `deep/Language.v`: Provides a shallow embedding of Boolean expressions and arithmetic expressions, and a deep embedding of our assertion language, on which we inductively define the meta operations of heap update and heap clear. We finally formalize Hoare triples and proof systems using weakest precondition and strongest postcondition axioms for the basic instructions.

- `deep/Classical.v`: Provides the classical semantics of assertions, and the strong partial correctness semantics of Hoare triples. Further it provides proofs of substitution lemmas corresponding to our meta-operators. Finally, it provides proofs of the soundness and completeness of the aforementioned proof systems.

D.2 Natural deduction

In this part of the artifact we use the classical axiom of excluded middle.

The proof system based on natural deduction is embedded in the Coq proof assistant using an axiomatic approach. It is known that adding axioms to Coq may affect soundness: it is future work to show that the axioms can be consistently used. Nonetheless, using the axiomatic approach allows us to stay close to the natural deduction proof system as introduced in this thesis.

```
Axiom D: Type.
Axiom hasval: D -> D -> Prop.
```

The type `D` is used as domain, over which we let the relation `hasval` range. This relation is the weak points to relation, so the strong points to relation can be defined in terms of this relation.

```
Axiom sep: Prop -> Prop -> Prop.
Axiom sepimp: Prop -> Prop -> Prop.
```

We also introduce axiomatically new connectives for use in propositions. Coq is also extended with syntax for constructing separation logic formulas.

```
Axiom rooted: Prop -> (D -> D -> Prop) -> Prop.
Axiom root_equiv: forall (A: Prop), A <-> rooted A hasval.
Axiom root_above: forall (A: Prop) h h',
  (forall x y, h x y <-> h' x y) -> rooted A h -> rooted A h'.
Axiom root_assoc: forall (A: Prop) h h',
  rooted (rooted A h) h' <-> rooted A (fun x y => rooted (h x y) h').
```

A rooted assertion consists of an assertion and a (first-order) description of the heap with respect to which it is evaluated. Note that in our axiomatization we do not limit that the given function actually is a first-order description. We also axiomatize that any assertion ϕ is equivalent to the rooted assertion $(\phi@\hookrightarrow)$, that we can replace equivalent descriptions of the heap, and extensionality of the description of the heap.

```
Axiom root_True: forall (h: D -> D -> Prop), rooted True h.
Axiom root_False: forall (h: D -> D -> Prop), rooted False h -> False.
Axiom root_hasval: forall (h: D -> D -> Prop) x y,
  rooted (hasval x y) h <-> h x y.
```

```
Axiom root_eq: forall (h: D -> D -> Prop) (T: Type) (x y: T),
  rooted (x = y) h <-> x = y.
Axiom root_split': forall (A B: Prop) h,
  (rooted A h /\ rooted B h) -> rooted (A /\ B) h.
Axiom root_join': forall (A B: Prop) h,
  (rooted A h \/ rooted B h) -> rooted (A \/ B) h.
Axiom root_and_elim: forall (A B: Prop) h,
  rooted (A /\ B) h -> rooted A h /\ rooted B h.
Axiom root_or_elim: forall (A B: Prop) h,
  rooted (A \/ B) h -> rooted A h \/ rooted B h.
Axiom root_imp': forall (A B: Prop) h,
  (rooted A h -> rooted B h) -> rooted (A -> B) h.
Axiom root_imp_elim': forall (A B: Prop) h,
  rooted (A -> B) h -> rooted A h -> rooted B h.
Axiom root_forall': forall (T: Type) (A: T -> Prop) h,
  (forall (x: T), rooted (A x) h) -> rooted (forall (x: T), A x) h.
Axiom root_forall_elim': forall (T: Type) (A: T -> Prop) h,
  rooted (forall (x: T), A x) h -> forall (x: T), rooted (A x) h.
Axiom root_exists': forall (T: Type) (A: T -> Prop) h,
  (exists (x: T), rooted (A x) h) -> rooted (exists (x: T), A x) h.
Axiom root_exists_elim': forall (T: Type) (A: T -> Prop) h,
  rooted (exists (x: T), A x) h -> exists (x: T), rooted (A x) h.
```

We also axiomatize reasoning about the classical connectives under a rooted assertion.

```
Definition Par (h1 h2: D -> D -> Prop) :=
  (forall x y, h1 x y -> forall z, ~h2 x z) /\
  (forall x y, h2 x y -> forall z, ~h1 x z).
Definition Split (h h1 h2: D -> D -> Prop) :=
  (forall x y, h x y <-> h1 x y \/ h2 x y) /\ Par h1 h2.
Axiom sep_elim': forall (A B C: Prop) (h: D -> D -> Prop),
  rooted (A ** B) h ->
  (forall h1 h2, Split h h1 h2 -> rooted A h1 -> rooted B h2 -> C) -> C.
Axiom sep_intro': forall (A B: Prop) (h: D -> D -> Prop),
  (exists h1 h2, Split h h1 h2 /\ rooted A h1 /\ rooted B h2) ->
  rooted (A ** B) h.
Axiom sepimp_elim': forall (A B C: Prop) (h h': D -> D -> Prop),
  rooted (A -** B) h ->
  Par h h' /\ rooted A h' /\
    (rooted B (fun x y => h x y \/ h' x y) -> C) -> C.
Axiom sepimp_intro': forall (A B: Prop) (h: D -> D -> Prop),
  (forall h', Par h h' -> rooted A h' ->
    rooted B (fun x y => h x y \/ h' x y)) ->
  rooted (A -** B) h.
```

Finally, we introduce axioms for reasoning about the new separating connectives. We use the syntax ** for separating conjunction and -** for separating implication.

From these axioms, it becomes possible to prove the following lemmas:

```
Lemma sep_assoc (A B C: Prop): (A ** B) ** C <-> A ** B ** C.
Lemma sep_Empty (A: Prop): A ** emp <-> A.
Lemma sep_or (A B C: Prop): (A \/ B) ** C <-> A ** C \/ B ** C.
Lemma sep_and (A B C: Prop): (A /\ B) ** C -> A ** C /\ B ** C.
Lemma adjoint (A B: Prop): A ** (A -** B) -> B.
```

Also the modalities can be defined and properties proven:

```
Definition box (A: Prop) := True ** (emp /\ (True -** A)).
Lemma box_elim (A: Prop): box A -> A.
Lemma box_indep (A: Prop): box A -> forall h, rooted (box A) h.
Lemma root_under: forall (A B: Prop),
  box (A -> B) -> forall h, rooted A h -> rooted B h.
Lemma box_rooted (A: Prop): (forall h, rooted A h) -> box A.
Lemma sep_mono (A A' B B': Prop):
  box (A -> A') -> box (B -> B') -> A ** B -> A' ** B'.
```

Finally, we can prove the equivalence:

```
Definition F1: Prop :=
  alloc x /\ ((x = y /\ z = w) \/ (x <> y /\ hasval y z)).
Definition F2: Prop :=
  pointsToDash x ** (pointsTo x w -** hasval y z).
Proposition F12': F1 -> F2.
Proposition F21: F2 -> F1.
```

The artifact consists of the following file:

- proof/Language.v: Provides an axiomatization of natural deduction for separation logic, and proves a number of lemmas based on theses axioms.

Bibliography

[1] Andrew Aberdein. Mathematical wit and mathematical cognition. *Topics in Cognitive Science*, 5(2):231–250, 2013.

[2] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In *International Workshop on Current Trends in Applied Formal Methods*, pages 168–183. Springer, 1998.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[4] Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic: To be or not to be created. In *2nd World Congress on Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*, pages 612–627. Springer, 2009.

[5] Miklós Ajtai. Isomorphism and higher order equivalence. *Annals of Mathematical Logic*, 16(3):181–203, 1979.

[6] Mahmudul Faisal Al Ameen. *Completeness of Verification System with Separation Logic for Recursive Procedures*. PhD thesis, Tokyo, 2016.

[7] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[8] Marc Andreessen. Why software is eating the world. *Wall Street Journal*, 2011.

[9] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*, volume 27 of *Applied Logic Series*. Springer, 2013.

[10] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs*. Springer, 3rd edition, 2009.

[11] Krzysztof R. Apt and Ernst-Rüdiger Olderog. Fifty years of Hoare's logic. *Formal Aspects of Computing*, 31(6):751–807, 2019.

[12] Lukas Armborst and Marieke Huisman. Permission-based verification of red-black trees and their merging. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 111–123. IEEE, 2021.

[13] Mario Rodriguez Artalejo. Some questions about expressiveness and relative completeness in Hoare's logic. *Theoretical computer science*, 39:189–206, 1985.

[14] Rob Arthan, Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. A general framework for sound and complete Floyd-Hoare logics. *ACM Transactions on Computational Logic (TOCL)*, 11(1):1–31, 2009.

[15] Callum Bannister, Peter Höfner, and Gerwin Klein. Backwards and forwards with separation logic. In Jeremy Avigad and Assia Mahboubi, editors, *9th International Conference on Interactive Theorem Proving (ITP)*, volume 10895 of *Lecture Notes in Computer Science*, pages 68–87. Springer, 2018.

[16] Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[17] Jon Barwise. *Handbook of mathematical logic*. Elsevier, 1982.

[18] Davide Basile. *Specification and Verification of Contract-Based Applications*. PhD thesis, University of Pisa, 2016.

[19] Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. Foundations for entailment checking in quantitative separation logic. In *31st European Symposium on Programming (ESOP)*, volume 13240 of *Lecture Notes in Computer Science*, pages 57–84. Springer, 2022.

[20] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2005.

[21] Jan A. Bergstra and John V. Tucker. Expressiveness and the completeness of Hoare's logic. *Journal of computer and system sciences*, 25(3):267–284, 1982.

[22] Jan A. Bergstra and John V. Tucker. Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs. *Theoretical Computer Science*, 17(3):303–315, 1982.

[23] Guram Bezhanishvili and Lawrence S. Moss. Undecidability of first-order logic, 2009. Educational module for the NSF-sponsored project on Learning Discrete Mathematics and Computer Science via Primary Historical Sources.

[24] Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, and Stijn de Gouw. Integrating ADTs in KeY and their application to history-based reasoning. In *24th International Symposium on Formal Methods (FM)*, volume 13047 of *Lecture Notes in Computer Science*. Springer, 2021.

[25] Leyla Bilge and Tudor Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security*, pages 833–844, 2012.

[26] David Binder, Thomas Piecha, and Peter Schroeder-Heister. *The Logical Writings of Karl Popper*. Springer, 2022.

[27] Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. In *10th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, volume 4423 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2007.

[28] Aleš Bizjak and Lars Birkedal. On models of higher-order separation logic. *Electronic Notes in Theoretical Computer Science*, 336:57–78, 2018.

[29] Frank S. de Boer and Hans-Dieter A. Hiep. Completeness and complexity of reasoning about call-by-value in Hoare logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(4), October 2021.

[30] M. Boogaard and E. Spoor. The software crisis in the Netherlands. In *Serie Research Memoranda*, No. 1994-21. Vrije Universiteit Amsterdam, 1994.

[31] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. Cambridge University Press, 2002.

[32] Richard Bornat. Proving pointer programs in Hoare logic. In *5th International Conference on Mathematics of Program Construction (MPC)*, pages 102–126. Springer, 2000.

[33] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 259–270, 2005.

[34] William D. Brewer. Gödel's doctoral thesis, 1928–30: The completeness of first-order logic. In *Kurt Gödel: The Genius of Metamathematics*, pages 101–129. Springer, 2022.

[35] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Information and Computation*, 211:106–137, 2012.

[36] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[37] Stephen Brookes. A revisionist history of concurrent separation logic. *Electronic Notes in Theoretical Computer Science*, 276:5–28, 2011.

[38] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

[39] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. *ACM SIGPLAN Notices*, 43(1):101–112, 2008.

[40] James Brotherston and Max Kanovich. Undecidability of propositional separation logic and its neighbours. In *25th IEEE Symposium on Logic in Computer Science (LICS)*, pages 130–139. IEEE, 2010.

[41] Luitzen Egbertus Jan Brouwer. Intuitionism and formalism. In A. Heyting, editor, *Philosophy and Foundations of Mathematics*, pages 123–138. Elsevier, 1975.

[42] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972.

[43] Cristiano Calcagno. *Semantic and Logical Properties of Stateful Programming.* PhD thesis, Universita di Genova, 2002.

[44] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, volume 3441 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2005.

[45] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. *Electronic Notes in Theoretical Computer Science*, 172:133–175, 2007.

[46] Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. Bringing order to the separation logic jungle. In *15th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 190–211. Springer, 2017.

[47] Chen Chung Chang and H. Jerome Keisler. *Model theory.* Elsevier, 1990.

[48] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *32nd ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2011.

[49] Krzysztof Ciesielski. *Set theory for the working mathematician.* Cambridge University Press, 1997.

[50] Edmund M. Clarke Jr. *Completeness and incompleteness theorems for Hoare-like axiom systems.* PhD thesis, Cornell University, 1976.

[51] Edmund M. Clarke Jr. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26(1):129–147, 1979.

[52] Edmund M. Clarke Jr. The characterization problem for Hoare logics. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522):423–440, 1984.

[53] Edmund M. Clarke Jr., Steven M. German, and Joseph Y. Halpern. Effective axiomatizations of Hoare logics. *Journal of the ACM*, 30(3):612–636, 1983.

[54] Timothy T.R. Colburn, James H. Fetzer, and R.L. Rankin. *Program verification: Fundamental issues in computer science*, volume 14 of *Studies in Cognitive Systems*. Springer, 2012.

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

[56] Stephen A. Cook and Derek C. Oppen. An assertion language for data structures. In *2nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 160–166, 1975.

[57] Laura Crosilla. Predicativity and Feferman. In Gerhard Jäger and Wilfried Sieg, editors, *Feferman on Foundations: Logic, mathematics, philosophy*, pages 423–447. Springer, 2017.

[58] H.-H. Dang, Peter Höfner, and Bernhard Möller. Algebraic separation logic. *Journal of Logic and Algebraic Programming*, 80(6):221–247, 2011.

[59] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound automation of magic wands. In *34th International Conference on Computer Aided Verification (CAV)*, volume 13372 of *Lecture Notes in Computer Science*, pages 130–151. Springer, 2022.

[60] Clayton Allen Davis, Onur Varol, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Botornot: A system to evaluate social bots. In *25th International Conference Companion on World Wide Web*, pages 273–274. ACM, 2016.

[61] Jacobus W. de Bakker. *Mathematical theory of program correctness*. Prentice-Hall, 1980.

[62] Jacobus W. de Bakker and Lambert G.L.T. Meertens. On the completeness of the inductive assertion method. *Journal of Computer and System Sciences*, 11(3):323–357, 1975.

[63] Frank S. de Boer. *Reasoning about Dynamically Evolving Process Structures; a proof theory for the parallel object-oriented language pool*. PhD thesis, Vrije Universiteit Amsterdam, April 1991.

[64] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDK's sort method for generic collections. *Journal of Automated Reasoning*, 62(1):93–126, 2019.

[65] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof Pearl: The KeY to correct and stable sorting. *Journal of Automated Reasoning*, 53(2):129–139, 2014.

[66] 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 *27th International Conference on Computer Aided Verification (CAV)*, volume 9206 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2015.

[67] Stéphane Demri and Morgan Deters. Logical investigations on separation logics. European Summer School on Logic, Language and Information (ESSLLI), 2015.

[68] Stéphane Demri and Morgan Deters. Separation logics and modalities: a survey. *Journal of Applied Non-Classical Logics*, 25(1):50–99, 2015.

[69] Stéphane Demri, Étienne Lozes, and Alessio Mansutti. The effects of adding reachability predicates in propositional separation logic. In *21st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 10803 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2018.

[70] Stéphane Demri, Étienne Lozes, and Alessio Mansutti. A complete axiomatisation for quantifier-free separation logic. *Logical Methods in Computer Science*, 17(3), 2021.

[71] Edsger W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[72] Yifan Ding, Nicholas Botzer, and Tim Weninger. Posthoc verification and the fallibility of the ground truth. *arXiv preprint arXiv:2106.07353*, 2021.

[73] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 287–300. ACM, 2013.

[74] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[75] Brijesh Dongol, Victor B.F. Gomes, and Georg Struth. A program construction and verification tool for separation logic. In *12th International Conference on Mathematics of Program Construction (MPC)*, volume 9129 of *Lecture Notes in Computer Science*, pages 137–158. Springer, 2015.

[76] Frédérick Douzet, Louis Pétiniaud, Loqman Salamatian, Kevin Limonier, Kavé Salamatian, and Thibaut Alchus. Measuring the fragmentation of the internet: the case of the Border Gateway Protocol (BGP) during the Ukrainian crisis. In *12th International Conference on Cyber Conflict (CyCon)*, volume 1300, pages 157–182. IEEE, 2020.

[77] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. On the expressive completeness of Bernays-Schönfinkel-Ramsey separation logic. *arXiv preprint arXiv:1802.00195*, 2018.

[78] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. The Bernays-Schönfinkel-Ramsey class of separation logic with uninterpreted predicates. *ACM Transactions on Computational Logic (TOCL)*, 21(3):1–46, 2020.

[79] Herbert B. Enderton. *A mathematical introduction to logic.* Elsevier, 2001.

[80] Gergö Érdi. *Compositional Type Checking.* Master thesis, Eötvös Loránd University, 2011.

[81] Mahmudul Faisal Al Ameen and Makoto Tatsuta. Completeness for recursive procedures in separation logic. *Theoretical Computer Science*, 631:73–96, 2016.

[82] William M. Farmer. *Simple Type Theory: A Practical Logic for Expressing and Reasoning About Mathematical Ideas.* Springer, 2023.

[83] Melvin Fitting. *Proof methods for modal and intuitionistic logics*, volume 169 of *Synthese Library.* Springer, 1983.

[84] Robert W. Floyd. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*, pages 65–81. Springer, 1993.

[85] Thomas Forster. Quine's New Foundations. In *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2019.

[86] Nissim Francez. *Program verification.* Addison-Wesley, 1992.

[87] Thomas Frayne, Anne Morel, and Dana Scott. Reduced direct products. *Fundamenta mathematicae*, 51(3):195–228, 1962.

[88] Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A Pérez. A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):841–869, 2022.

[89] Didier Galmiche and Dominique Larchey-Wendling. Expressivity properties of Boolean BI through relational models. In *26th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2006.

[90] Didier Galmiche and Daniel Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2010.

[91] Mohan Ganesalingam. *The language of mathematics.* Springer, 2013.

[92] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In *25th International Conference on Formal Methods for Industrial Critical Systems (FMICS)*, volume 12327 of *Lecture Notes in Computer Science*, pages 3–69. Springer, 2020.

[93] Kurt Gödel. *Über die vollständigkeit des logikkalküls.* PhD thesis, University of Vienna, 1929.

[94] Michael D. Godfrey and David F. Hendry. The computer as Von Neumann planned it. *IEEE Annals of the History of Computing*, 15(1):11–21, 1993.

[95] Joseph A. Goguen et al. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, 1997.

[96] Gerald Arthur Gorelick. *A complete axiomatic system for proving assertions about recursive and non-recursive programs.* Master thesis, University of Toronto, 1975.

[97] Clemens Grabmayer. *Relating proof systems for recursive types.* PhD thesis, Vrije Universiteit Amsterdam, 2005.

[98] Clemens Grabmayer. From abstract rewriting systems to abstract proof systems. *arXiv preprint arXiv:0911.1412*, 2009.

[99] Michal Grabowski. On relative completeness of Hoare logics. *Information and control*, 66(1-2):29–44, 1985.

[100] David Gries. *The science of programming.* Springer, 2012.

[101] Jan Friso Groote, Ammar Osaiweran, and Jacco H. Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 467–472. IEEE, 2011.

[102] Reiner Hähnle. Dijkstra's legacy on program verification. In C.A.R. Hoare Krzysztof R. Apt, editor, *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 105–140. ACM, 2022.

[103] Anthony Hall. Seven myths of formal methods. *IEEE software*, 7(5):11–19, 1990.

[104] Anthony Hall. Realising the benefits of formal methods. In *7th International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *Lecture Notes in Computer Science*. Springer, 2005.

[105] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[106] Richard Wesley Hamming. The unreasonable effectiveness of mathematics. *The American Mathematical Monthly*, 87(2):81–90, 1980.

[107] David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer, 1979.

[108] Leon Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 14(3):159–166, 1949.

[109] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2), 1950.

[110] Andreas Herzig. A simple separation logic. In *20th International Workshop on Logic, Language, Information, and Computation (WoLLIC)*, volume 8071 of *Lecture Notes in Computer Science*, pages 168–178. Springer, 2013.

[111] Joel Hestness, Stephen W. Keckler, and David A. Wood. A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–160. IEEE, 2014.

[112] Hans-Dieter A. Hiep. *New Foundations for Separation Logic (Coq artifact)*, 2024. `https://dx.doi.org/10.5281/zenodo.10558424`.

[113] Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. History-based specification and verification of Java collections in KeY. In *16th International Conference on Integrated Formal Methods (iFM)*, volume 12546 of *Lecture Notes in Computer Science*, pages 199–217. Springer, 2020.

[114] Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. Verifying OpenJDK's LinkedList using KeY (extended paper). *International Journal on Software Tools for Technology Transfer*, 24(5):783–802, 2022.

[115] Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, and Stijn de Gouw. Verifying OpenJDK's LinkedList using KeY. In *26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12079 of *Lecture Notes in Computer Science*, pages 217–234. Springer, 2020.

[116] Scott A. Hissam, Daniel Plakosh, and C. Weinstock. Trust and vulnerability in open source software. *IEEE Proceedings-Software*, 149(1):47–51, 2002.

[117] Charles Anthony Richard Hoare. How did software get so reliable without proof? In *3rd International Symposium of Formal Methods Europe (FME)*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[118] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[119] Wilfrid Hodges. *A shorter model theory.* Cambridge University Press, 1997.

[120] Johannes Hostert, Andrej Dudenhefner, and Dominik Kirst. Undecidability of dyadic first-order logic in Coq. In *13th International Conference on Interactive Theorem Proving (ITP)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[121] Zhé Hóu and Alwen Tiu. Completeness for a first-order abstract separation logic. In *14th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 10017 of *Lecture Notes in Computer Science*, pages 444–463. Springer, 2016.

[122] Zhe Hou and Alwen Tiu. Completeness for a first-order abstract separation logic. In Atsushi Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 444–463, 2016.

[123] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal methods: From academia to industrial practice. *arXiv preprint arXiv:2002.07279*, 2020.

[124] Roberto Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer languages*, 19(1):19–40, 1993.

[125] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, 2001.

[126] Sushil Jajodia, Paulo Shakarian, V.S. Subrahmanian, Vipin Swarup, and Cliff Wang. *Cyber Warfare: Building the Scientific Foundation*, volume 56 of *Advances in Information Security.* Springer, 2015.

[127] Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *18th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 226–234, 1997.

[128] Capers Jones. Measuring defect potentials and defect removal efficiency. *Journal of Defense Software Engineering*, 21(6):11–13, 2008.

[129] Capers Jones and Olivier Bonsignour. *The economics of software quality.* Addison-Wesley Professional, 2011.

[130] Paul C. Jorgensen and Byron DeVries. *Software testing: a craftsman's approach.* CRC Press, 5th edition, 2021.

[131] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of the ACM on Programming Languages*, volume 2 of *POPL*, pages 1–34. ACM, 2017.

[132] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[133] Samuel Kamin. The expressive theory of stacks. *Acta informatica*, 24:695–709, 1987.

[134] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, et al. A Coq library for mechanised first-order logic. In *The Coq Workshop 2022*. hal.science, 2022.

[135] Stephen Cole Kleene. *Introduction to metamathematics*. Wolters-Noordhoff, 1971.

[136] Stephen Cole Kleene. The work of Kurt Gödel. *Journal of Symbolic Logic*, 41(4):761–778, 1976.

[137] Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[138] Tomasz Kowaltowski. *Correctness of programs manipulating data structures*. PhD thesis, University of California, Berkeley, 1973.

[139] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, 2017.

[140] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *4th International Workshop on Types in Language Design and Implementation*, pages 105–116. ACM, 2009.

[141] Krishan Kumar and Sonal Dahiya. Programming languages: A survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5(5):307–313, 2017.

[142] Mark Steven Laventhal. *Verification of programs operating on structured data*. Bachelor and master thesis, MIT, 1974.

[143] Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. *ACM SIGPLAN Notices*, 49(1):477–490, 2014.

[144] Martti Lehto and Pekka Neittaanmäki. *Cyber security: Critical infrastructure protection*, volume 56 of *Computational Methods in Applied Sciences*. Springer, 2022.

[145] Ewen Maclean, Andrew Ireland, and Gudmund Grov. Proof automation for functional correctness in separation logic. *Journal of Logic and Computation*, 26(2):641–675, 2016.

[146] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[147] Makarov Evgeny Maratovich. Dynamic separation logic and its use in education. *Современные информационные технологии и ИТ-образование*, 16(3):543–550, 2020.

[148] Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Information and Media Technologies*, 4(2):304–316, 2009.

[149] Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. Hoare logic in the abstract. In *20th International Workshop on Computer Science Logic (CSL)*, volume 4207 of *Lecture Notes in Computer Science*, pages 501–515. Springer, 2006.

[150] Jeferson Martínez and Javier M. Durán. Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study. *International Journal of Safety and Security Engineering*, 11(5):537–545, 2021.

[151] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2006.

[152] Tom F. Melham. *Higher order logic and hardware verification.* Cambridge University Press, 2009.

[153] Elliott Mendelson. *Introduction to Mathematical Logic.* CRC Press, 6th edition, 2015.

[154] Bertrand Meyer. Design by contract and the component revolution. In *34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS).* IEEE Computer Society, 2000.

[155] Ronald Middelkoop. *A proof system for object oriented programming using separation logic.* Master thesis, Technische Universiteit Eindhoven, 2003.

[156] Raúl E. Monti, Robert Rubbens, and Marieke Huisman. On deductive verification of an industrial concurrent software component with VerCors. In *International Symposium on Leveraging Applications of Formal Methods*, pages 517–534. Springer, 2022.

[157] J.H. Morris. Verification oriented language design. Technical report, University of California, Berkeley, 1972.

[158] Joseph M. Morris. A general axiom of assignment. *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by F.L. Bauer, E.W. Dijkstra and C.A.R. Hoare*, pages 25–34, 1982.

[159] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[160] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The art of software testing*. Wiley Online Library, 2nd edition, 2004.

[161] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *23rd European Symposium on Programming Languages and Systems (ESOP)*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer, 2014.

[162] Russell O'Connor. Classical mathematics for a constructive world. *Mathematical Structures in Computer Science*, 21(4):861–882, 2011.

[163] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1):271–307, 2007.

[164] Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). *Software safety and security*, 33:286–318, 2012.

[165] Peter W. O'Hearn. Incorrectness logic. In *Proceedings of the ACM on Programming Languages*, volume 4 of *POPL*, pages 1–32. ACM, 2019.

[166] Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[167] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999.

[168] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *15th International Workshop on Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[169] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In *7th ACM Symposium on Theory of Computing*, pages 107–116, 1975.

[170] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

[171] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[172] Jens Pagel and Florian Zuleger. Strong-separation logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–40, 2022.

[173] Matthew J. Parkinson. Local reasoning for Java. Technical report, University of Cambridge, Computer Laboratory, 2005.

[174] David Lorge Parnas. Really rethinking 'formal methods'. *Computer*, 43(1):28–34, 2010.

[175] David Parsons. *Foundational Java: Key Elements and Practical Programming.* Springer, 2020.

[176] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 2884 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2003.

[177] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789. Springer, 2013.

[178] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In *26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 711–728. Springer, 2014.

[179] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science*, pages 109–121. IEEE, 1976.

[180] David Pym, Jonathan M. Spring, and Peter W. O'Hearn. Why separation logic works. *Philosophy & Technology*, 32:483–516, 2019.

[181] David J. Pym. The semantics and proof theory of the logic of bunched implications. In *Applied Logic Series*, 2002.

[182] Panu Raatikainen. Gödel's incompleteness theorems. In *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2020.

[183] Brian Randell. The 1968/69 NATO software engineering reports, 1996.

[184] Habib ur Rehman, Eiad Yafi, Mohammed Nazir, and Khurram Mustafa. Security assurance against cybercrime ransomware. In *International Conference on Intelligent Computing & Optimization (ICO)*, pages 21–34. Springer, 2018.

[185] Andrew Reynolds, Radu Iosif, and Cristina Serban. Reasoning in the Bernays-Schönfinkel-Ramsey fragment of separation logic. In *18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *Lecture Notes in Computer Science*, pages 462–482. Springer, 2017.

[186] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in smt. In *International Symposium on Automated Technology for Verification and Analysis*, pages 244–261. Springer, 2016.

[187] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, Cornerstones of Computing, pages 303–321. Macmillan Education UK, 2000.

[188] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[189] John C. Reynolds. An overview of separation logic. In *First Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 4171 of *Lecture Notes in Computer Science*, pages 460–469. Springer, 2005.

[190] John C. Reynolds. An introduction to separation logic. In *Engineering Methods and Tools for Software Safety and Security*, pages 285–310. IOS Press, 2009.

[191] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005.

[192] Dennis M. Ritchie. The development of the C programming language. In Richard G. Gibson Thomas J. Bergin, editor, *History of Programming languages*, pages 671–698. ACM, 1996.

[193] David S. Rosenblum. Formal methods and testing: why the state-of-the art is not the state-of-the practice. *ACM SIGSOFT Software Engineering Notes*, 21(4):64–66, 1996.

[194] Yaman Roumani. Patching zero-day vulnerabilities: an empirical analysis. *Journal of Cybersecurity*, 7(1), 2021.

[195] Mohammad Salahuddin, Khorshed Alam, and Ilhan Ozturk. The effects of Internet usage and economic growth on CO_2 emissions in OECD countries: A panel investigation. *Renewable and Sustainable Energy Reviews*, 62:1226–1235, 2016.

[196] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.

[197] Ravi Sen. Challenges to cybersecurity: Current state of affairs. *Communications of the Association for Information Systems*, 43(1):2, 2018.

[198] Syed Muhammad Ali Shah, Maurizio Morisio, and Marco Torchiano. An overview of software defect density: A scoping study. In *19th Asia-Pacific Software Engineering Conference*, volume 1, pages 406–415. IEEE, 2012.

[199] Stewart Shapiro. *Foundations without foundationalism: A case for second-order logic.* Clarendon Press, 1991.

[200] Sajjan G. Shiva. *Advanced computer architectures.* CRC Press, 2018.

[201] Mihaela Sighireanu, Juan A Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, et al. Sl-comp: competition of solvers for separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 116–132. Springer, 2019.

[202] Raymond M. Smullyan. *Gödel's incompleteness theorems.* Oxford University Press, 1992.

[203] Raymond M. Smullyan. *First-Order Logic.* Dover, 1995.

[204] James Somers. The coming software apocalypse. *The Atlantic*, 26:1, 2017.

[205] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[206] Bjarne Stroustrup. A history of C++ 1979–1991. In Richard G. Gibson Thomas J. Bergin, editor, *History of Programming languages*, pages 671–698. ACM, 1996.

[207] Johanna Stuber. *Verification of Red-Black Trees in KeY: A Case Study in Deductive Java Verification.* Bachelor thesis, Karlsruher Institut für Technologie (KIT), 2023.

[208] Andy S. Tatman. *Analysis and Formal Specification of OpenJDK's BitSet.* Bachelors thesis, Leiden University, 2023.

[209] Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. Completeness and expressiveness of pointer program verification by separation logic. *Information and Computation*, 267:1–27, 2019.

[210] Balder ten Cate, Johan van Benthem, and Jouko Vaananen. Lindström theorems for fragments of first-order logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 280–292. IEEE, 2007.

[211] Aditya Thakur, Jason Breck, and Thomas Reps. Satisfiability modulo abstraction for separation logic with linked lists. In *International SPIN Symposium on Model Checking of Software*, pages 58–67. ACM, 2014.

[212] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2nd edition, 2000.

[213] Anne Sjerp Troelstra and Dirk Van Dalen. *Constructivism in Mathematics, Vol 1*. Elsevier, 1988.

[214] Anne Sjerp Troelstra and Dirk Van Dalen. *Constructivism in Mathematics, Vol 2*. Elsevier, 1988.

[215] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–108, 2007.

[216] Jouko Väänänen. Second order logic or set theory? *Bulletin of Symbolic Logic*, 18(1):91–121, 2012.

[217] Johan van Benthem and Kees Doets. Higher-order logic. In *Handbook of philosophical logic*, pages 189–243. Springer, 1983.

[218] Guido van Rossum and Fred L. Drake Jr. Python tutorial. Technical report, Centrum voor Wiskunde en Informatica, 1995.

[219] Moshe Y. Vardi. Move fast and break things. *Communications of the ACM*, 61(9):7–7, 2018.

[220] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[221] Benjamin Wagner. Understanding internet shutdowns: A case study from Pakistan. *International Journal of Communication*, 12(1):22, 2018.

[222] Tjark Weber. Towards mechanized program verification with separation logic. In *18th International Workshop on Computer Science Logic (CSL)*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2004.

[223] Mark Allen Weiss. *Data structures and algorithm analysis in Java*. Pearson Education, 2012.

[224] Stephen B. Wicker. The ethics of zero-day exploits—the NSA meets the trolley car. *Communications of the ACM*, 64(1):97–103, 2020.

[225] Eugene P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. *Communications on Pure and Applied Mathematics*, 13:1–14, 1960.

[226] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[227] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, 1993.

[228] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In *33rd International Conference on Computer Aided Verification (CAV)*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

[229] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *6th International ACM Symposium on Machine Programming*, pages 1–10. ACM, 2022.

[230] Hongseok Yang. *Local reasoning for stateful programs.* PhD thesis, University of Illinois, 2001.

[231] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[232] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2002.

[233] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*, pages 274–283. IEEE, 2009.

[234] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[235] Job Zwiers, Ulrich Hannemann, Yassine Lakhnech, Willem P. de Roever, and Frank A. Stomp. Modular completeness: Integrating the reuse of specified software in top-down program development. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 595–608. Springer, 1996.

List of Publications

1. *Dynamic separation logic*
   Frank S. de Boer, Hans-Dieter A. Hiep, Stijn de Gouw
   In: *Proceedings of MFPS XXXIX*
   Electronic Notes in Theoretical Informatics and Computer Science, volume 3
   Episciences, 2023

2. *Formal Specification and Analysis of OpenJDK's BitSet Class*
   Andy S. Tatman, Hans-Dieter A. Hiep, Stijn de Gouw
   In: *iFM 2023: 18th International Conference, iFM 2023, Proceedings*
   Lecture Notes in Computer Science, volume 14300
   Springer, 2023

3. *The logic of separation logic: models and proofs*
   Frank S. de Boer, Hans-Dieter A. Hiep, Stijn de Gouw
   In: *Automated Reasoning with Analytic Tableaux and Related Methods: 32nd International Conference, TABLEAUX 2023, Proceedings*
   Lecture Notes in Computer Science, volume 14278
   Springer, 2023

4. *Integrating ADTs in KeY and their application to History-Based Reasoning about Collection*
   Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, Stijn de Gouw
   Formal Methods in System Design, volume 61
   Springer, 2022

5. *Footprint logic for object-oriented components*
   Frank S. de Boer, Stijn de Gouw, Hans-Dieter A. Hiep, Jinting Bian
   In: *Formal Aspects of Component Software: 18th International Conference, FACS 2022, Proceedings*
   Lecture Notes in Computer Science, volume 13712
   Springer, 2022

6. *Verifying OpenJDK's LinkedList using KeY (extended paper)*
   Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Stijn de Gouw
   International Journal on Software Tools for Technology Transfer, volume 24
   Springer, 2022

7. *Integrating ADTs in KeY and their application to History-Based Reasoning*
   Jinting Bian, Hans-Dieter A. Hiep, Frank S. de Boer, Stijn de Gouw
   In: *Formal Methods, 24th International Symposium, FM 2021, Proceedings*
   Lecture Notes in Computer Science, volume 13047
   Springer, 2021

8. *Completeness and complexity of reasoning about call-by-value in Hoare logic*
   Frank S. de Boer, Hans-Dieter A. Hiep
   In: *ACM Transactions On Programming Languages And Systems*
   Volume 43, Issue 4
   Association for Computing Machinery, 2021

9. *History-Based Specification and Verification of Java Collections in KeY*
   Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, Stijn de Gouw
   In: *Integrated Formal Methods, 16th International Conference, IFM 2020, Proceedings*
   Lecture Notes in Computer Science, volume 12546
   Springer, 2020

10. *A Tutorial on Verifying LinkedList Using KeY*
    Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, Stijn de Gouw
    In: *Deductive Software Verification: Future Perspectives, Reflections on the Occasion of 20 Years of KeY*
    Lecture Notes in Computer Science, volume 12345
    Springer, 2020

11. *History-based specification and verification of Java collections in KeY*
    Frank S. de Boer, Hans-Dieter A. Hiep
    In: *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs*
    Association for Computing Machinery, 2020

12. *Reowolf: Synchronous Multi-party Communication over the Internet*
    Christopher Esterhuyse, Hans-Dieter A. Hiep
    In: *Formal Aspects of Component Software, 16th International Conference, FACS 2019, Proceedings*
    Lecture Notes in Computer Science, volume 12018
    Springer, 2019

13. *Verifying OpenJDK's LinkedList using KeY*
    Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, Marko van Eekelen, Stijn de Gouw
    In: *Tools and Algorithms for the Construction and Analysis of Systems, 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Proceedings, Part II*
    Lecture Notes in Computer Science, volume 12079
    Springer, 2020

14. *Reowolf 1.0: Project Documentation*
    Christopher A. Esterhuyse, Hans-Dieter A. Hiep
    Technical Report
    CWI, 2020

15. *Axiomatic Characterization of Trace Reachability for Concurrent Objects*
    Frank S. de Boer, Hans-Dieter A. Hiep
    In: *Integrated Formal Methods, 15th International Conference, IFM 2019, Proceedings*
    Lecture Notes in Computer Science, volume 11918
    Springer, 2019

See also the ORCiD page (0000-0001-9677-6644) for the current list of publications.