Vorlesungsskript *Rechnergestütztes Beweisen*

Martin Hofmann

WS 2003/04

1 Introduction

Computer-aided theorem proving means to carry out mathematical proofs on a computer whose job it is to check steps, to perform bookkeeping tasks and to automate routine steps. Conducting a proof on a computer may be compared to and has a lot in common with implementing an informally given algorithm or model. For example, a number of details must be filled in and, more importantly, mistakes and shortcomings of the high-level model are brought to the surface.

Computer-aided theorem proving has numerous applications in program and hardware verification as well as prototype development. To a lesser, perhaps increasing, degree it is used to aid the development of genuine mathematical proofs.

1.1 Course outline

In this course, we will get to know the computer-based theorem prover PVS (`pvs.csl.sri.com`) along with its theoretical foundations and some ramifications thereof. Those are for instance

- Logical foundations: sequent calculus, set theory
- Expressive power: cut elimination
- Automation of equational reasoning: rewriting
- Automation of logical reasoning: resolution
- Modularisation and structuring: type theory

- Finite state verification: Modal logics and model checking
- Infinite state verification: Abstract model checking
- Independent checking of proof certificates: type theory

Mostly in the tutorials we will apply this knowledge to a variety of problems from

- Solving logical puzzles
- Algorithms on lists and trees
- Hardware components such as adder, counter, multiplier
- Distributed algorithms using invariants and reasoning
- Distributed algorithms using abstract model checking
- Experiments with other theorem provers.

1.2 Notions of proof

What exactly is a "proof". When asked this question a typical mathematician would produce something like the following: A proof is a convincing, undebatable argument establishing the truth of a mathematical statement. Back to Euclid (300 B.C.) goes the following concretion of this definition: A proof is a derivation of a statement from axioms by means of logical rules.

This sound good, but it remains to say what "axioms" should be and what the "logical rules" are. In Euclid's case (geometry) the axioms were truisms such as "for any two non-equal points there is exactly one line passing through them". The logical rules were essentially the ones we still use today and will learn about later in the course. An example of such a rule: if "A implies B" holds and "A" holds then "B" holds, too (*modus ponens*).

Later on, more complicated concepts such as real numbers and limits were introduced which made it less clear what reasonable axioms should be. For example, even the famous 18th century mathematician Leonhard Euler struggled with the infinite series $1 - 1 + 1 - 1 + 1 - 1 + \dots$ and ended up ascribing the value $1/2$ to it on the basis of the same informal mathematical reasoning he used for his celebrated theorems.

The lack of solid logical foundations for mathematics, and in particular analysis led to an actual crisis in mathematics (*Grundlagenkrise*) which was settled early in the last century by the invention of set theory (and, following up on this, by the formalisation of real numbers, limits, integrals and so on by Weierstrass, Riemann and others.)

1.2.1 Set theory

Set theory is a formalism which allows one to *define* all other mathematical concepts and to *prove* their axioms, thus enabling a rigorous proof of their *consistency*, i.e., sensibility. For instance, we can define points as triples of real numbers (which in turn are defined as certain sequences of rational numbers (which in turn are defined as certain pairs of integer numbers (which in turn are defined as certain pairs of natural numbers (which are defined as certain sets: $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, etc.)))) and then *prove* that through any two distinct points goes one and only one line (a line being defined, for instance, as a set of points satisfying some linear relation).

The present formulation of set theory consists of approximately nine axioms[1] among them

- two sets having the same elements are equal
- for each set we can form the set of its subsets
- there exists an infinite set
- for each set we can form the subset consisting of those elements sharing a given property

Using a formalised language statements in set theory can be written as strings and recognised as such, for example the above four axioms are written as follows:

- $\forall a.\forall b.(\forall x.x \in a \iff x \in b) \Rightarrow a = b$
- $\forall a.\exists b.\forall x.x \in b \iff x \subseteq a$
  (where $x \subseteq a \stackrel{\text{def}}{=} \forall y.y \in x \Rightarrow y \in a$)

[1] the precise number depends on what we count as axiom and what as a logical rule.

- $\exists a.\neg\text{finite}(a)$
  (where $\text{finite}(a) \stackrel{\text{def}}{=} \forall b.(\emptyset \in b \wedge (\forall c.\forall x.c \in b \Rightarrow c \cup \{x\} \in b)) \Rightarrow a \in b)$
  Here $c \cup \{x\}$ is a notation for a set whose existence is asserted by two other axioms (union and singleton).

- $\forall a.\exists b..\forall x.x \in b \Leftrightarrow (x \in a \wedge \phi(x))$ (where $\phi(x)$ is an arbitrary statement involving x)

We should remark at this point that despite the formal notation the axioms of set theory remain unproved and their justification relies on philosophical and pragmatic arguments.

The *logical rules* of set theory are precisely the ones of first-order logic which we are going to learn about in more detail later in the course.

1.2.2 Proofs as formal derivations

Once we have a formal concept of axioms and rules, we can define a *proof* of a statement ϕ as a sequence of statements

$$\phi_0, \phi_1, \phi_2, \phi_3, \ldots, \phi_n = \phi$$

ending in ϕ such that each ϕ_i is either an axiom or follows from previous statements by a logical rule.

Sequence of statements

$$\phi_0, \phi_1, \phi_2, \phi_3, \ldots, \phi_n$$

such that each ϕ_i is an axiom or an immediate consequence by a logical rule of $\{\phi_0, \ldots, \phi_{i-1}\}$

So, to check whether an alleged proof indeed is one is a matter of entirely mechanical symbol manipulation and does not require any creative skills or intelligence.

In practice, however, writing out proofs at this level of detail would be far too cumbersome and so checking whether a purported proof, say in a journal submission, indeed is one, does require considerable mathematical skill and devotion!

And while the vast majority of mathematicians agrees that any proof in mathematics can *theoretically* be formalised in set theory and hence mechanically checked, many of them believe that *in practice* such formalisation is impossible for all but the simplest toy examples. This belief might have remained unchallenged if there had not been the request for formalised proofs from informatics and the advent of sufficiently powerful machines.

1.3 Proof assistants

So, why do we need formalised proofs in informatics? Well, the correctness of software (or hardware) is nothing but a mathematical statement amenable to formalisation and mechanical checking.

Here are some examples of formal theorem proving occurs in informatics.

- Is program X correct? *very rare*
- Does method X satisfy invariant Y?
- Does variable X always hold values with property Y? E.g. Y=points to a sorted linked list, a balanced binary tree, data items consistent with store.
- Does protocol X guarantee property Y? E.g. Y=cache coherence, sequential execution, absence of deadlock, ...)
- Does circuit X implement function Y? E.g. Y=FP multiplication, Fourier transform,
- Does algorithm X satisfy specification Y? E.g. X=garbage collector, Y=absence of interference+liveness.
- Does Theorem X about programming language Y hold? E.g. X= type safety, correctness of proof rules.
- Verification of certificates ("proof-carrying code")

While in early stages of soft- and hardware development (actually until the early nineties, I would say) these proofs could easily be carried out by hand (possibly using some notation and intermediate calculations) the size of systems has reached a state where this has become impossible in many cases.

Prompted by these requirements systems called *proof assistants* or *theorem provers* have been developed which perform not only the task of checking sizeable formalised proofs but also help with coming up with formalised proofs in the first place by bookkeeping assumptions and variables, providing tactics and decision procedures (e.g. for propositional formulas, certain fragments of arithmetic, modal and temporal logic, equational theories, etc.) and by providing libraries of definitions and already proved theorems.

- Bookkeeping of assumptions and variables
- Tactics
- Decision procedures
- Libraries of definitions and theorems

Figure 1: Tasks of a proof assistant

- Expressive logic
- Powerful decision procedures
- Large body of basic notions

Figure 2: Strengths of PVS

1.3.1 The PVS System

In this course we will get to know one such proof assistant in some detail, namely the PVS system developed by Owre, Rushby, and Shankar at SRI, Menlo Park, California. As any system, PVS has a number of strengths and also weaknesses. As a partial compensation for the weaknesses we will later in the course take a look at complementary systems such as SPASS, Coq, Isabelle.

PVS has a very expressive underlying logic (classical higher-order logic), it comes equipped with a number of powerful decision procedures, e.g., for linear arithmetic and equational reasoning, and it has a large body of basic notions which allow one to start a formalisation on a relatively high level.

On the other hand, PVS has recurrent soundness problems, that is, from time to time someone finds out that a weird combination of tactics use and language features allows one to prove $0 = 1$! Moreover, there is no formal representation of proofs. One reason why these problems are not trivial to fix is precisely the large body of basic notions which here turns into a disadvantage.

1.3.2 Soundness and proof objects

A problem with a proof assistant is that its correct behaviour is hard to verify. Whether a word processor provides decent looking output can be checked at a

- Soundness problems
- No formal representation of proofs
- Large body of basic notions
- (Bad heuristics for first-order instantiation)

Figure 3: Weaknesses of PVS

```
 |-------
[1]   FORALL (y: t, v_106: list[t]):
        (FORALL (x: t):
           occ(x, merge(null, v_106)) =
                 occ(x, null) + occ(x, v_106))
         IMPLIES
         (FORALL (x: t):
            occ(x, merge(null, cons(y, v_106))) =
             occ(x, null) + occ(x, cons(y, v_106)))
Rule? (grind)
```

lots of rewrites etc. are printed

```
This completes the proof of merge1.4.
```

Figure 4: A quick proof in PVS

glance, similarly a video game either is fun to play with or not.

On the other hand, correct behaviour of a proof assistant is rather hard to detect. After all it's because we don't want to do the proofs by hand that we use a proof assistant in the first place. The "output" of a proof assistant doesn't consist of a nice looking document or a thrilling sequence of images.

In PVS you can have a complicated looking subgoal to prove, you type in `(grind)` and PVS responds that this proves the statement. There is no way to check this proof independently; all that's being recorded is that the tactic `(grind)` has been invoked.

PVS stands for *prototype verification system* which is explained by the following quote from the PVS Prover Guide 2.3, see `pvs.csl.sri.com`.

> *The primary purpose of PVS is to provide formal support for conceptualization and debugging in early stages of the life cycle of a hardware or software system. In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable. We find that the best way to analyse such an abstract specification is by attempting proofs of desirable consequences of the specification.*

So, provided soundness problems occur rarely [2] they do not really compromise the usability of the system.

Proofs as guarantee In recent years researchers have proposed a use of proofs as a certificate not unlike the cryptographic certificates such as digital signatures, etc. While the latter certify authenticity of a datum, i.e., a relationship between the datum and the sender, a formal proof certifies a property of the datum itself which is independent of the sender.

For example, a third-party provider of a component of a safety-critical system might be required to provide a formal proof of correctness.

A referee of a paper in mathematics or Theoretical Informatics might not be willing to verify all details of a proof but would rather run the formalised proofs of the theorems in the paper through a proof checker.

Also it has been proposed (under the name "proof-carrying code") that ...

The design of systems like Coq (to some extent also Isabelle) is such that independent verification is possible. These systems generate a formal representation of a proof (a *proof object* or *proof term*) amenable to separate verification by a *proof checker* which is simple and small. Even if tactics or decision procedures contain bugs these will always show up at the checking stage so the worst that can happen is that an attempted proof has to be redone.

At present it seems that none of the systems with explicit proof objects can compete with PVS or similar systems. However, I think that this is mainly a problem of organisation and manpower, not an inherent theoretical one. I believe that in the not too distant future we will see powerful proof assistants with (almost) bug-free proof-checkers inside.

[2] they do sometimes, see the PVS web site

Atoms: $A, B, C, D, \ldots$
Connectives:
$\phi \wedge \psi$: ϕ "and" ψ (conjunction)
$\phi \vee \psi$: ϕ "or" ψ (disjunction)
$\phi \Rightarrow \psi$: ϕ "implies" ψ (implication)
$\neg\phi$: "not" ϕ (negation)

Precedence: $\neg > \wedge > \vee > \Rightarrow$
Example: $(A \Rightarrow B) \wedge \neg A \Rightarrow \neg A$ reads $((A \Rightarrow B) \wedge (\neg A)) \Rightarrow (\neg A)$

Figure 5: Syntax of propositional formulas

2 Sequent calculus

In this section we will learn about the "logical rules" which underly the PVS system. In their present form they were introduced by the logician Gerhard Gentzen around 1940 as a means to analyse the proof-theoretic strength of formal arithmetic.

2.1 Formulas

We start with a set of *atoms*, aka *identifiers* or *symbols* $A, B, C, D, \ldots$. *Formulas* are built up from atoms by the *connectives* $\vee, \wedge, \Rightarrow$ (binary) and $\neg$ (unary), so $(A \Rightarrow B) \wedge \neg A \Rightarrow C$ is a formula. By convention the binding power is $\neg > \wedge > \vee > \Rightarrow$, so the above formula reads $((A \Rightarrow B) \wedge (\neg A)) \Rightarrow C$.

The *meaning* of a formula is given relative to an interpretation of the atoms as either true or false. For instance, if A is true, and B, C are both false, then our example formula will be true because then $A \Rightarrow B$ is false (the only way for an implication to be false is that its antecedent (here A) is true and its consequent (here B) is false).

Digression on semantics of implication Please notice that this so-called *classical interpretation of implication* is sometimes at odds with our intuitive understanding of implication. For instance, the sentence "if MH wears a tie during the lecture then he can turn lead into gold." is actually true under this interpretation. It is possible to formalise the intuitive meaning of such sentence by implicitly quantifying over a set of *worlds* that describe possibilities. One could then imagine a world in which MH wears ties and also worlds in which he can turn lead into gold, If $\eta(A) = \mathsf{tt}, \eta(B) = \eta(C) = \mathsf{ff}$ then

$$\begin{array}{ll} & \eta((A \Rightarrow B) \wedge \neg A \Rightarrow C) \\ = & \eta(A \Rightarrow B) \wedge \eta(\neg A) \Rightarrow \mathsf{ff} \\ = & (\mathsf{ff} \Rightarrow \mathsf{tt}) \wedge \mathsf{ff} \Rightarrow \mathsf{ff} = \mathsf{tt} \end{array}$$

Figure 6: Formal meaning of a formula

$$A \Rightarrow A$$

$$((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

$$\begin{array}{l} \neg((A_{11} \vee A_{12}) \wedge (A_{21} \vee A_{22}) \wedge (A_{31} \vee A_{32}) \wedge \\ \quad \neg(A_{11} \wedge A_{21}) \wedge \neg(A_{11} \wedge A_{31}) \wedge \neg(A_{21} \wedge A_{31}) \wedge \\ \quad \neg(A_{12} \wedge A_{22}) \wedge \neg(A_{12} \wedge A_{32}) \wedge \neg(A_{22} \wedge A_{32})) \end{array}$$

Figure 7: Examples of tautologies

but the latter would not form a superset of the former because there is no causal relationship whatsoever. On the other hand, for $\phi \Rightarrow \psi$ to be valid in this refined sense one requires that the set of worlds in which ψ holds forms a superset of the set of worlds in which ϕ holds.

Formalisation of meaning Anyway, under the aforementioned classical interpretation the formula $\phi \stackrel{\text{def}}{=} (A \Rightarrow B) \wedge \neg A$ is true only if A is false hence $\phi \Rightarrow \neg A$ comes out true no matter what the atoms actually stand for.

More formally, a partial function η mapping atoms to $\{\mathsf{tt}, \mathsf{ff}\}$ can be extended to formulas by interpreting the connectives in the obvious way.

A formula is a *tautology* if its meaning is true regardless of the interpretation of the atoms.

A formula is *satisfiable* if it is true for *some* interpretation of the atoms. Clearly, A is satisfiable if and only if $\neg A$ is not a tautology and therefore A is a tautology if and only if $\neg A$ is unsatisfiable.

- if Mary likes champaign then either Bob or Alice like red wine...
- Planning problems: find sequence of actions for a robot to—say—remove a certain item from a stockpile
- Behaviour of digital hardware circuits
- Combinatorial optimisation (scheduling, routing,...)
- ...

Figure 8: Applications of propositional logic

2.2 Applications of propositional logic

Many naturally occurring problems admit encodings in propositional logic in the sense that to know whether a certain formula is satisfiable or a tautology amounts to having a solution to the problem at hand. Examples are summarised in Figure 2.2.

It is therefore an important practical problem to determine whether a given propositional formula is a tautology or not and (equivalently) whether or not it is satisfiable.

A formula with 100 atoms admits ca. 10^{33} different valuations so checking tautologies by examining truth tables may be unfeasible.

While no method is known to date which would be inherently better than checking truth tables there has been considerable progress in the last years at solving instances arising from practical problems (SAT solvers). These solvers vastly outperform any human logician trying to attack propositional formulas by logical reasoning! So why should we look at axioms and logical rules for propositional calculus?

The answer is that in many situations the atoms will themselves be complex formulas, typically a defined predicate applied to some variables, such as $x \in a$ or $\mathtt{sorted}(\mathbf{list}_1)$ and we want to be able to break down the validity of a formula involving these atoms into basic implications between them.

This is precisely the goal of sequent calculus which we will now describe.

Sequents: $\Gamma \Longrightarrow \Delta$ where $\Gamma = \phi_1, \ldots, \phi_m$ and $\Delta = \psi_1, \ldots, \psi_n$ are lists of formulas.
Meaning: $\phi_1 \wedge \cdots \wedge \phi_m \Rightarrow \psi_1 \vee \cdots \vee \psi_n$
Examples: $A \Rightarrow B, C \Rightarrow D \Longrightarrow U \wedge (\neg V \wedge B)$
$A \Rightarrow B, A \Longrightarrow B$
$\Longrightarrow A, A \Rightarrow B$
$A, \neg A \Longrightarrow$
$\Longrightarrow$

Figure 9: Syntax of sequents

2.3 Sequents

A *sequent* is an expression of the form $\Gamma \Longrightarrow \Delta$ where Γ, Δ are (possibly) empty lists of formulas.

The *meaning* of a sequent $\Gamma \Longrightarrow \Delta$ is defined as the meaning of the formula $\bigwedge \Gamma \Rightarrow \bigvee \Delta$ where $\bigwedge \Gamma$ is the conjunction ("and", $\wedge$) of the formulas in Γ and $\bigvee \Delta$ is the disjunction ("or", $\vee$) of the formulas in Δ.

For example, our formula $(A \Rightarrow B) \wedge \neg A \Rightarrow C$ is equivalent to the sequent $A \Rightarrow B, \neg A \Longrightarrow C$.

A *proof* in sequent calculus is an upside-down tree labelled with sequents such that the leaves are labelled with *axioms* and the label of an internal node is the conclusion of a *rule* which has the labels of its immediate descendants as premises.

A sequent $\Gamma \Longrightarrow \Delta$ is an *axiom* if Γ and Δ have a formula in common. For example, the sequent $A, B \Longrightarrow A, C$ is an axiom.

The *rules* for sequent calculus are best read backwards, i.e. "what do I need to prove in order to establish a sequent $\Gamma \Longrightarrow \Delta$?". First, we have *structural rules* allowing to permute, duplicate or remove formulas. In the literature sequents are often defined as pairs of *sets* rather than lists of formulas. This makes all the structural rules except WEAK redundant. For computer-aided formal reasoning it is, however, useful to have explicit access to formulas, say by their position in a list.

For each connective there are two *logical rules*, one when the connective appears on the left, and one when it appears on the right.

To understand, e.g., $\vee$-L try to think as follows: to prove Δ under the assumption $\phi \vee \psi$ (and some other stuff Γ) we must make a case distinction as to whether ϕ or ψ holds, hence we must prove Δ under assumptions Γ, ϕ and then again under

$$\frac{\Gamma_1, \phi, \psi, \Gamma_2 \Longrightarrow \Delta}{\Gamma_1, \psi, \phi, \Gamma_2 \Longrightarrow \Delta} \quad \text{(PERM-L)}$$

$$\frac{\Gamma \Longrightarrow \Delta_1, \phi, \psi, \Delta_2}{\Gamma \Longrightarrow \Delta_1, \psi, \phi, \Delta_2} \quad \text{(PERM-R)}$$

$$\frac{\Gamma \Longrightarrow \Delta}{\Gamma, \phi \Longrightarrow \Delta} \quad \text{(WEAK-L)}$$

$$\frac{\Gamma \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta, \phi} \quad \text{(WEAK-R)}$$

$$\frac{\Gamma, \phi, \phi \Longrightarrow \Delta}{\Gamma, \phi \Longrightarrow \Delta} \quad \text{(CONTR-L)}$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi, \phi}{\Gamma \Longrightarrow \Delta, \phi} \quad \text{(CONTR-R)}$$

Figure 10: Structural rules

assumptions Γ, ψ.

Rule $\Rightarrow$-R is probably the easiest of these: to prove $\phi \Rightarrow \psi$ we must prove ψ under the additional assumption ϕ (if we disregard the side formulas $\Gamma, \Delta \ldots$). If we forget about Δ we can also explain $\neg$-R: to prove $\neg\phi$ we must derive a contradiction (empty Δ) from the assumption ϕ.

The $\neg$-L rule says: if $\neg\phi$ is among our assumptions then to prove anything (Δ) its enough to prove ϕ (or Δ straightaway, of course). This is known as *ex falso quodlibet*.

Rule $\Rightarrow$-L, finally, says: if we want to use the assumption $\phi \Rightarrow \psi$ then we can add its conclusion (ψ) to our assumptions provided we succeed (independently) in proving its antecedent (ϕ).

One can derive the implication rules from the encoding of $\phi \Rightarrow \psi$ as $\neg\phi \vee \psi$.

2.4 Soundness and completeness

Definition: A sequent is *derivable* if there exists a proof with it as root label.

Theorem: A sequent is derivable if and only if it is a tautology.

Proof: Let us call a *proof tree* a tree whose nodes (and leaves) are labelled

$$\frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta} \quad (\wedge\text{-L})$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi \qquad \Gamma \Longrightarrow \Delta, \psi}{\Gamma \Longrightarrow \Delta, \phi \wedge \psi} \quad (\wedge\text{-R})$$

$$\frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi \Longrightarrow \Delta} \quad (\vee\text{-L})$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi, \psi}{\Gamma \Longrightarrow \Delta, \phi \vee \psi} \quad (\vee\text{-R})$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi}{\Gamma, \neg\phi \Longrightarrow \Delta} \quad (\neg\text{-L})$$

$$\frac{\Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta, \neg\phi} \quad (\neg\text{-R})$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \Rightarrow \psi \Longrightarrow \Delta} \quad (\Rightarrow\text{-L})$$

$$\frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \Delta, \phi \Rightarrow \psi} \quad (\Rightarrow\text{-R})$$

$$\dfrac{\dfrac{\dfrac{A \Longrightarrow B, A \quad B, A \Longrightarrow B}{A \Rightarrow B, A \Longrightarrow B}\Rightarrow\text{-L}}{A \Rightarrow B \Longrightarrow B, \neg A}\neg\text{-L}}{A \Rightarrow B, \neg B \Longrightarrow \neg A}\neg\text{-L}$$

Figure 11: Example proof

$$\dfrac{\dfrac{A, B \Longrightarrow A, A \wedge C \quad A, B \Longrightarrow B, A \wedge C}{A, B \Longrightarrow A \wedge B, A \wedge C} \quad \dfrac{A, C \Longrightarrow A \wedge B, A \quad A, C \Longrightarrow A \wedge B, C}{A, C \Longrightarrow A \wedge B, A \wedge C}}{A, (B \vee C) \Longrightarrow A \wedge B, A \wedge C}$$

Figure 12: Example proof

with sequents in such a way that whenever a node labelled S has immediate ancestors $S_1, \ldots, S_n$ then there is a logical rule with $S_1, \ldots, S_n$ as assumptions as S as conclusion. For example, $S_1 = A \Longrightarrow B$ and $S_2 = A \Longrightarrow C$ and $S = A \Longrightarrow B \wedge C$. Given the form of our rules we always have $n = 1$ or $n = 2$. The leaves may but do not need to be labelled with axioms. Let us write $S_1, \ldots S_\ell \vdash S$ to mean that there is a proof tree whose root is labelled S and whose leaves are labelled with $S_1, \ldots S_\ell$.

Notice that a *proof* of sequent S is a proof tree all whose leaves are labelled with axioms.

By induction on (depth of) proof trees one easily shows that if $S_1, \ldots, S_n \vdash S$ and $S_1, \ldots, S_n$ are all true under some valuation η then S, too, comes out true under η. Recall that the truth value of a sequent $S = \phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots \psi_n$ under some valuation η is defined as $\bigwedge_i \eta(\phi_i) \Rightarrow \bigvee_j \eta(\psi_j)$, i.e., S comes out false precisely if all the ϕ_i come out true and all the ψ_j come out false.

The above proves that if a sequent has a proof, i.e., a proof tree with axioms labelling its leaves, then it is tautologous, i.e., true under all valuations.

For completeness we first notice (again by induction on proof trees) that if $S_1, \ldots, S_n \vdash S$ is proved by a proof tree not involving rule WEAK then S is equivalent to the conjunction of the S_i, i.e., S comes out false under some valuation η as soon as one of the S_i is falsified by η. Now, for any sequent S we can always find a proof tree not involving rule WEAK whose root is labelled S and whose leaves are labelled with sequents consisting of atoms only. This is done by successively "breaking down" all the connectives in S. If S is *not derivable* then at least one of the atomic sequents labelling the leaves of this proof tree will not be an axiom (otherwise our proof tree would be a proof!). This sequent will thus be of the form $A_1, \ldots A_m \Longrightarrow B_1, \ldots B_n$ where the A_i and B_j are atoms and $\{A_1, \ldots, A_m\} \cap \{B_1, \ldots, B_n\} = \emptyset$. Any valuation η with $\eta(A_i) = \mathtt{tt}, \eta(B_j) = \mathtt{ff}$ will falsify this sequent, hence S. So S is not a tautology. □

We remark that this also shows that if a sequent is derivable with rules CONTR and WEAK then it is provable without those rules; the "generic" proof tree obtained by breaking down the connectives must lead to a proof in this case, otherwise we would obtain a falsifying valuation.

2.4.1 Linear logic

The fact that rules WEAK and CONTR can be eliminated is due to their being built into the other rules and axioms. In *linear logic* weakening and contraction are removed and the side formulas in different premises of a rule are required to be disjoint. For instance, a linear version of $\wedge$-R would look thus:

$$\frac{\Gamma_1 \Longrightarrow \Delta_1, \phi \qquad \Gamma_2 \Longrightarrow \Delta_2, \psi}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2, \phi \wedge \psi} \qquad (\wedge\text{-R-}\textsc{Lin})$$

Moreover, the only axioms are $A \Longrightarrow A$ for A an atom. If we replace all rules and axioms by their linear versions and remove WEAK and CONTR (which now are no longer redundant) we obtain a (a fragment of) linear logic in which, e.g., the formula $A \Rightarrow A \wedge A$ is not provable.

2.4.2 Cut elimination

In spite of completeness it is useful to have yet another rule: the famous cut rule:

$$\frac{\Gamma_1 \Longrightarrow \Delta_1, \phi \qquad \Gamma_2, \phi \Longrightarrow \Delta_2}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2} \qquad (\textsc{Cut})$$

Since CUT obviously transforms tautologies into tautologies ...

Usually, Γ_1 and Δ_1 are empty. In this case, CUT corresponds to invocation of a lemma: if we have proved ϕ "as a lemma" then we can add it to our list of assumptions whenever we want. The theorem says that lemmas can always be eliminated.

3 Introduction to PVS

As already mentioned, PVS is based on the sequent calculus. We will start by using PVS as a proof-assistant for that system.

To do that we call PVS from the (Linux!) command line with

```
pvs
```

This brings up an Emacs window entitled PVS. A possible question concerning context creation should be answered affirmatively (means: type in `yes`).

Now create a file named `sequent.pvs` containing something like

```
sequent_calculus: THEORY
BEGIN

A,B,C,D: VAR boolean
A11,A12,A21,A22,A31,A32: VAR boolean
```

```
K : PROPOSITION
    A IMPLIES B IMPLIES A

S : PROPOSITION
    (A IMPLIES B IMPLIES C) IMPLIES (A IMPLIES B)
        IMPLIES  (A  IMPLIES C)

Peirce : PROPOSITION
    ((A IMPLIES B) IMPLIES A) IMPLIES A

Contra : PROPOSITION
     (A IMPLIES B) IMPLIES NOT B IMPLIES NOT A

dist1 : PROPOSITION
    A AND (B OR C) IMPLIES (A AND B) OR (A AND C)

dist2 : PROPOSITION
    A AND B OR C AND D IMPLIES (A OR C) AND  (B OR D)

schub : PROPOSITION
    NOT (
     (A11 OR A12) AND
     (A21 OR A22) AND
     (A31 OR A32) AND
     NOT (A11 AND A21) AND
     NOT (A11 AND A31) AND
     NOT (A21 AND A31) AND
     NOT (A12 AND A22) AND
     NOT (A12 AND A32) AND
     NOT (A22 AND A32))

END sequent_calculus
```

Click on the first "proposition". This will bring up a prover window and a prompt `Rule?`. The first rule you should enter is `(skolem!)`. This will get rid of the `FORALL` quantifier which we'll talk about later and basically display the formula as a sequent with empty premise list and one conclusion named `[1]`. Unfortunately, all the atoms are decorated with `!1`. ¡ Now you can

- apply a disjunctive rule ($\wedge$-L, $\vee$-R, $\Rightarrow$-R) by entering `(flatten` x`)` where x is the number of the formula you want to apply the rule to. (x must obviously be 1 at the beginning.
- apply a conjunctive rule ($\wedge$-R, $\vee$-L, $\Rightarrow$-L) by entering `(split` x`)` where, again, x is the number of the formula.

The rules for negation are applied automatically.

Try to prove all the propositions in this way but don't waste too much time on the last one (`schub`).

Rather give up after a few steps by typing `(quit)` and redo the proof (answering no when asked whether you want to rerun the existing proof) this time typing `(prop)` after the `(skolem!)` step or `(grind)` right at the start.

You can display the tree structure of your current proof with `M-x x-show-current-proof` and of a finished proof with `M-x x-show-proof`.

You may wish to do some more ad-hoc experiments with PVS. Extensive documentation is available at the PVS homepage `pvs.csl.sri.com`.

4 Resolution

Gentzen's sequent calculus provides a decision procedure for the validity of propositional formulas: construct the proof tree as in the completeness proof and check whether all leaves are labelled with axioms.

Unfortunately, the complexity of this procedure is exponential in the size of the formula to be proved. This is due to the duplication of "goals" in rules $\vee$-L, $\wedge$-R, $\Rightarrow$-L. Worse still, a lot of work is done twice: if we break down $\Gamma \Longrightarrow \Delta, \phi \wedge \psi$ into $\Gamma \Longrightarrow \Delta, \phi$ and $\Gamma \Longrightarrow \Delta, \psi$ then the "breaking down" of Γ and Δ must be done in each branch individually.

Of course, unless $P = NP$ we cannot expect a really efficient (polynomial) method for deciding propositional formulas; however, there are algorithms that behave quite well in practice for moderately sized formulas. One of these is the method of *resolution* invented by ROBINSON which we will take a look at for one thing for its popularity and for another because its applicability to first-order logic which we will come to shortly.

Resolution is a method for deciding *satisfiability* of a propositional formula presented as a *set of clauses*. Recall that a formula ϕ is satisfiable if $\neg\phi$ is not a tautology. A *literal* is either an atom or a negated atom, e.g., $A, \neg B, \mathrm{door\_closed}, \neg\mathrm{alarm\_on}$ are all literals. A *clause* is a set of literals whose meaning is their disjunction.

- a *literal* is either an atom or a negated atom: $A, \neg B$, ¬door_closed.
- a *clause* is a set of literals understood as their *disjunction*: $\{\neg\text{lift\_moves}, \text{door\_closed}, \text{alarm\_on}\}$
- a *set of clauses* is understood as the conjunction of the individual clauses.
- empty clause = ff, empty set of clauses = tt
- One is interested in *satisfiability* of sets of clauses.
- Validity (to be a tautology) is trivial for sets of clauses. Why?

Figure 13: Clauses

Some people write a clause as an explicit disjunction using $\vee$, others use set notation. A *set of clauses* represents the conjunction of the individual clauses. Watch out for the empty clause (representing ff) and the empty set of clauses (representing tt).

To check whether a formula ϕ is a tautology we can represent $\neg\phi$ as a set of clauses and see whether it is not satisfiable.

4.1 Representation of formulas as sets of clauses

We all know that any formula ϕ can be converted into conjunctive normal form, by "multyplying out" according to de Morgan's rule. The problem with this is that in general the size of the conjunctive normal form will be exponential in the size of the formula to start with. Actually, if this blow up would not occur we had a simple method for checking whether ϕ is a tautology. Just bring ϕ into conjunctive normal form and see whether each clauses is a tautology.

What we can do without exponential blowup, though, is to construct a set of clauses $\mathcal{C}$ which is *satisfiable* iff ϕ is. To that end, we proceed as follows: first, we may assume that ϕ contains connectives $\vee, \wedge, \neg$ only and that, moreover, $\neg$ occurs in front of atoms only. One calls this the negation normal form of ϕ. Now, if ϕ happens to be a literal then there is nothing to do. If $\phi = \phi_1 \wedge \phi_2$ and $\phi_1, \phi 2$ are equi-satisfiable with $\mathcal{C}_1, \mathcal{C}_2$, respectively, then $\mathcal{C}_1 \cup \mathcal{C}_2$ is equi-satisifable with ϕ. If, finally, $\phi = \phi_1 \vee \phi_2$ then $(\mathcal{C}_1 \vee P) \cup (\mathcal{C}_2 \vee \neg P$ is equi-satisfiable with ϕ. Here P is a fresh atom and $\mathcal{C} \vee P$ means the addition of P to each clause in $\mathcal{C}$.

Rule of resolution:
$$\frac{C_1 \cup \{\ell\} \qquad C_2 \cup \{\neg\ell\}}{C_1 \cup C_2}$$
For example: $\{A, \neg B, C\}$ and $\{\neg A, \neg B, \neg C\}$ yield $\{A, \neg A, \neg B\}$ and then $\{\neg B\}$.
$\{A, \neg A\}$ yields $\{A, \neg A\}$
$\{A\}, \{\neg A\}$ yields $\emptyset$ (empty clause)

Figure 14: The rule of resolution

INPUT: a set of clauses $\mathcal{C}$
WHILE $\emptyset \notin \mathcal{C}$ OR $\mathcal{C}$ still grows DO
  CHOOSE $C_1, C_2 \in \mathcal{C}$ S. T. $\ell \in C_1$ and $\neg\ell \in C_2$ for some literal ℓ.
  $\mathcal{C} := \mathcal{C} \cup \{C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}\}$
IF $\emptyset \in \mathcal{C}$ THEN OUTPUT "$\mathcal{C}$ is not satisfiable"
  ELSE OUTPUT "$\mathcal{C}$ is satisfiable"

Figure 15: The method of resolution

4.2 The method of resolution

The method of resolution decides whether a given set of clauses is satisfiable. It works as follows: given two clauses C_1 and C_2 such that C_1 contains some literal ℓ and C_2 contains its negation $\neg\ell$ (with the understanding that the negation of $\neg A$ is A) then the *rule of resolution* applied to C_1, C_2 yields the clause $C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}$. For example, applying the rule of resolution to $\{A, \neg B, C\}$ and $\{\neg A, \neg B, \neg C\}$ yields $\{A, \neg A, \neg B\}$. Applying the rule of resolution to the resulting clause $C = C_1 = C_2$ yields $\{\neg B\}$.

The method of resolution consists of closing up a set of clauses under the rule of resolution.

4.3 Correctness of resolution

If the set of clauses—after this closure—contains the empty clause then it is unsatisfiable, otherwise we can find a satisfying valuation. This is the content of the correctness theorem for resolution.

Theorem: Let $\mathcal{C}$ be a set of clauses closed under resolution, i.e., the result of applying the rule of resolution to any two clauses in $\mathcal{C}$ is already contained in $\mathcal{C}$. Then $\mathcal{C}$ is satisfiable iff $\emptyset \notin \mathcal{C}$.

Proof: If a set of clauses $\mathcal{C}'$ has been obtained from a *satisfiable* set of clauses $\mathcal{C}$ by a single application of the rule of resolution then $\mathcal{C}'$ is satisfiable, too. One says that the rule of resolution preserves satisfiability. Thus, if we derive from $\mathcal{C}$ a set of clauses containing the empty clauses then $\mathcal{C}$ must have been unsatisfiable.

For the converse, suppose that the set of clauses $\mathcal{C}$ is closed under the rule of resolution, i.e., any application of the rule of resolution to clauses in $\mathcal{C}$ yields a clause that is already contained in $\mathcal{C}$.

We claim that if $\mathcal{C}$ does not contain the empty clause then $\mathcal{C}$ is satisfiable. We prove the claim by induction on the number of variables. The case of zero atoms is clear, so suppose that $\mathcal{C}$ contains some atom A. We define the clause sets $\mathcal{C}[A \mapsto \mathtt{tt}]$ by removing from $\mathcal{C}$ every clause that contains the literal A and removing the literal $\neg A$ from every clause that contains it. Analogously, we define $\mathcal{C}[A \mapsto \mathtt{ff}]$.

For example, $\{\{A\}, \{\neg A\}\}[A \mapsto \mathtt{tt}] = \{\emptyset\}$ (unsatisfiable) yet $\{\{A, \neg B\}\}[A \mapsto \mathtt{tt}] = \{\} = \emptyset$ (satisfiable).

It is easily seen by case distinction that these sets are closed under the rule of resolution, too, so the induction hypothesis applies to them. Now, if both $\mathcal{C}[A \mapsto \mathtt{tt}]$ and $\mathcal{C}[A \mapsto \mathtt{ff}]$ contain the empty clause so does $\mathcal{C}$ so suppose w.l.o.g. that $\emptyset \notin \mathcal{C}[A \mapsto \mathtt{tt}]$. Inductively, we find η satisfying $\mathcal{C}[A \mapsto \mathtt{tt}]$. The valuation $\eta[A \mapsto \mathtt{tt}]$ then satisfies $\mathcal{C}$.

4.4 Implementation issues

Having established the correctness of the above abstract nondeterministic algorithm we can try to think of deterministic concretisations all of which will automatically be correct since our proof worked for the abstract nondeterministic version, i.e., no matter how we choose clauses for resolution and how we detect that our clause set does not grow anymore.

An aside: the question as to when one program *refines* another one in the sense that whenever the former enjoys certain correctness properties so does the latter, is a topic of still ongoing research. More precisely, the questions are what should qualify as a "correctness property" and how one can ensure refinement without explicitly checking out all those correctness properties.

Anyway, to refine the above resolution algorithm one will for example maintain some array of pairs of clauses that have not yet been resolved with each other and choose candidates for resolution from that array removing them thereafter. Ties can be broken by aiming for short clauses. Another popular improvement is the amalgamation of several resolution steps into a single one (*hyperresolution*).

4.5 Long resolution proofs

While for many tautologies (or rather their negations) resolution works astonishingly fast there are other ones, e.g., `schub` above for which it is rather slow. Indeed, HAKEN has shown that the resolution method has exponential worst case complexity.

Theorem (Haken): There is a constant $c > 1$ and an infinite family of (unsatisfiable) sets of clauses $P_1, P_2, \ldots$ such that P_n consists of $O(n^2)$ clauses yet any derivation of $\emptyset$ from P_n will involve $O(c^n)$ many clauses.

To wit, the set of clauses P_n expresses that $n + 1$ pigeons fit into n holes, e.g., $P_2 \Leftrightarrow \neg\texttt{schub}$.

The proof of Haken's theorem is elementary but fairly long and technical. Recently, WIGDERSON has presented an important simplification. Use Google to find his paper if you are interested.

4.6 Cook's programme

Haken's theorem is quite drastic evidence for exponential time complexity of the resolution procedure. Even if Haken's theorem would not hold as stated resolution could still fail to be polynomial time: it might be difficult to find a polynomially sized derivation of $\emptyset$ even if it exists and in the case of *satisfiable* sets of clauses the process of closing up might result in exponentially many clauses.

It is an important open complexity-theoretic question related to P=NP as to whether there is a proof system for propositional logic with the property that any tautology ϕ has a proof of size polynomial in the size of ϕ. Refuting this for concrete proof systems (as Haken has done for resolution) is a popular research activity initiated by S. COOK. As far as I know it is presently not known whether or not each tautology has a polynomially sized proof in sequent calculus with the CUT rule.

5 First-order logic

Propositional calculus is nice, but in many applications we need a way of talking about elements, predicates, and operations. That is what first order logic is for. Figure 5 shows some examples of sentences that lend themselves to a formalisation in first-order logic. A formal system for writing down such statements is obtained by augmenting propositional logic with the *quantifiers* $\forall$ (for all) and $\exists$

1. Every student has a matric number.
2. If a student fails to matriculate she will be expelled.
3. Every human being is a philosopher.
4. There always is one student who complains about every course.
5. There is a set with no elements.
6. For every natural number n there exists a natural number d such that $2^d \leq n$ and $n < 2^{d+1}$.
7. The knirp of each bilg is a prugl but does not bebelf any quist.

Figure 16: First-order formulas (informal)

(there exists). As to the range of these quantifiers one has two options which we consider in order.

Untyped first-order logic. This is the traditional and most common version of first-order logic. We let all quantifiers and variables range over one and the same implicit, a priori given, domain. In this case we must use special predicates to restrict the range of quantifiers. Figure 17 shows how the example sentences are formalised in untyped first-order logic.

Typed first-order logic. Alternatively, we fix a collection of *types* and require that each quantifier is annotated with a type determining its range. Assuming that we have fixed types

human, student, number, course, set, bilg, quist

we could write the example sentences as in Figure 5. Untyped first-order logic has the advantage of being slightly simpler to formulate and it suffices for many applications, especially in mathematics. Typed first-order logic has the advantage of being more readable and, more importantly, that it allows the user to distinguish between actual properties he wants to prove and typing judgments which should follow automatically in most cases. For example, if we were to prove formula 6 above in the untyped setting then we would at some point come up with a s and

1. $\forall x.$student$(x) \Rightarrow \exists n.$number$(x) \wedge$ has_matric_no(x, n)
2. $\forall x.$student$(x) \Rightarrow \neg$has_matriculated$(x) \Rightarrow$ will_be_expelled(x)
3. $\forall x.$human$(x) \Rightarrow$ philosopher(x)
4. $\exists x.$student$(x) \wedge \forall c.$course$(c) \Rightarrow$ complains_about(x, c)
5. $\exists x.\forall y.\neg x \in y$
6. $\forall n.$number$(n) \Rightarrow \exists d.$number$(d) \wedge$ leq(power$(2, d), n) \wedge$ lt$(n,$ power$(2,$ plus$(d, 1)))$
7. $\forall x.$bilg$(x) \Rightarrow$ prugl(knirp$(x)) \wedge \forall y.$quist$(y) \Rightarrow \neg$bebelfs(knirp$(x), y)$

Figure 17: Formalisation in untyped first-order logic

1. $\forall x$:student.$\exists n$:number.has_matric_no(x, n)
2. $\forall x$:student $\Rightarrow \neg$has_matriculated$(x) \Rightarrow$ will_be_expelled(x)
3. $\forall x$:human.philosopher(x)
4. $\exists x$:student.$\forall c$:course.complains_about(x, c)
5. $\exists x$:set.$\forall y.\neg y \in x$
6. $\forall n$:number.$\exists d$:number.leq(power$(2, d), n)\wedge$ lt(power$(2,$ plus$(d, 1)))$
7. $\forall x$:bilg.prugl(knirp$(x)) \wedge \forall y$:quist.$\neg$bebelfs(knirp$(x), y)$

Figure 18: Typed first-order logic

would then have to prove number(d) as well as the actually interesting property of d.

In the typed setting the first one falls under typing and can often be discharged automatically.

Of course, the distinction between types and predicates is a subjective one and can be "misused".

Typechecking can be difficult. For example, if D is the "type" consisting of quadruples of integers (x, y, z, n) such that $n > 2$ and $x^n + y^n = z^n$ then proving

$$\exists p{:}D.p = p$$

is tantamount to proving Wiles' theorem!

If used reasonably then types can considerably simplify (formal) proofs and the appearance of statements.

5.1 Typed first-order language

A typed first-order language is specified by the following data:

1. a collection $\mathcal{T}$ of *types*
2. a collection $\mathcal{P}$ of *predicate constants*, each endowed with an *arity* $[\tau_1, \ldots, \tau_n \to \text{boolean}]$ where $\tau_1, \ldots, \tau_n \in \mathcal{T}$
3. a collection $\mathcal{F}$ of function constants each endowed with an arity $[\tau_1, \ldots, \tau_n \to \tau_{n+1}]$ where $\tau_1, \ldots, \tau_{n+1} \in \mathcal{T}$

The arity of a predicate constant is nothing but a list of types; the brackets, the arrow, and "boolean" are merely notation. We use the notation $P : [\tau_1, \ldots, \tau_n \to \text{boolean}]$ and $f : [\tau_1, \ldots, \tau_n \to \tau_{n+1}]$ to indicate the arities of predicate, resp. function constants.

Examples: For the formulas 1,. . . ,4 an appropriate language is as follows:

$$\begin{array}{rcl}
\mathcal{T} & = & \{\text{student}, \text{number}, \text{course}\} \\
\mathcal{P} & = & \{\text{has_matric_no} : [\text{Student}, \text{number} \to \text{boolean}] \\
 & & \text{has_matriculated} : [\text{student} \to \text{boolean}] \\
 & & \text{will_be_expelled} : [\text{student} \to \text{boolean}]]] \\
 & & \text{complains_about} : [\text{student}, \text{course} \to \text{boolean}] \\
\mathcal{F} & = & \emptyset
\end{array}$$

Typing contexts: $K = x_1{:}\tau_1, \dots, x_n{:}\tau_n$; formally: finite function from variables to types.
Typing rules:

$$\frac{K(x) = \tau}{K \triangleright_{\mathcal{L}} x : \tau} \qquad \text{(Var)}$$

$$\frac{f : [\tau_1, \dots, \tau_n \to \tau_{n+1}] \in \mathcal{F} \qquad K \triangleright_{\mathcal{L}} t_1 : \tau_1, \dots, K \triangleright_{\mathcal{L}} t_n : \tau_n}{K \triangleright_{\mathcal{L}} f(t_1, \dots, t_n) : \tau_{n+1}} \qquad \text{(Fun)}$$

Figure 19: Well formed terms

For the formula 7 an appropriate first-order language would be

$$\begin{array}{lcl} \mathcal{T} & = & \{\text{bilg}, \text{quist}, \text{knirp\_t}\} \\ \mathcal{P} & = & \{\text{prugl} : [\text{knirp\_t} \to \text{boolean}], \text{bebelfs} : [\text{knirp\_t}, \text{quist} \to \text{boolean}]\} \\ \mathcal{F} & = & \{\text{knirp} : [\text{bilg} \to \text{knirp\_t}] \end{array}$$

The type knirp_t arises only as the range of a function symbol not as the range of a quantifier. When translating this formula from natural language to formal language we could also have opted for a predicate constant knirp : [bilg, knirp_t $\to$ boolean] denoting its graph. The content of the definite article "*the* knirp of…" could be rendered by another formula stating unique existence. This would require the notion of equality which we will come to later.

5.2 Syntax

In order to define properly what formulas are we have to talk about terms and formulas possibly involving variables as those occur in scopes of quantifiers. We thus assume an infinite set $\mathcal{V}$ of variables distinct from the other symbols and fix a first order language $\mathcal{L}$. A *typing context* is a finite partial function K mapping variables to types. If K is a typing context and $x \notin \mathrm{dom}(K)$ and $\tau \in \mathcal{T}$ then $K, x{:}\tau$ is the typing context K extended with $x \mapsto \tau$.

We write $K \triangleright_{\mathcal{L}} t : \tau$ to mean that t is a well formed term in $\mathcal{L}$ of type τ possibly involving the variables declared and typed as given by K. This judgment is inductively defined by the following *typing rules*.

Well-formed formulas are built up from atomic formulas (predicate constants applied to appropriately typed terms) by propositional connectives and quantifiers which *bind* variables. Formally, we introduce the judgment $K \triangleright_{\mathcal{L}} \phi$: boolean to mean that ϕ is a well formed formula in $\mathcal{L}$ possibly involving the free variables declared and typed in K. This judgment is defined by the following rules:

$$\frac{P : [\tau_1, \ldots, \tau_n \to \text{boolean}] \in \mathcal{P} \qquad K \rhd_{\mathcal{L}} t_1 : \tau_1, \ldots, K \rhd_{\mathcal{L}} t_n : \tau_n}{K \rhd_{\mathcal{L}} P(t_1, \ldots, t_n) : \text{boolean}} \quad (\textsc{Atom})$$

$$\frac{K \rhd_{\mathcal{L}} \phi : \text{boolean}}{K \rhd_{\mathcal{L}} \neg\phi : \text{boolean}} \quad (\textsc{Neg})$$

$$\frac{K \rhd_{\mathcal{L}} \phi : \text{boolean} \qquad K \rhd_{\mathcal{L}} \psi : \text{boolean} \qquad \star \in \{\vee, \wedge, \Rightarrow\}}{K \rhd_{\mathcal{L}} \phi \star \psi : \text{boolean}} \quad (\textsc{Conn})$$

$$\frac{K, x{:}\tau \rhd_{\mathcal{L}} \phi : \text{boolean} \qquad Q \in \{\forall, \exists\}}{K \rhd_{\mathcal{L}} Qx{:}\tau.\phi : \text{boolean}} \quad (\textsc{Quant})$$

These rules define *abstract syntax* together with *typing* (there latter is also known and misnamed as “semantic analysis”). For concrete syntax one needs to specify precedence rules and use parentheses to disambiguate otherwise. The connectives take precedence as before; quantifiers always extend as far to the right as possible, i.e., until an unmatched closing parenthesis is encountered.

5.3 Semantics

A formula ϕ is *closed* if it contains no free variables, i.e., if $\emptyset \rhd_{\mathcal{L}} \phi$: boolean. These are the ones we are really interested in; the open formulas are introduced only as an auxiliary device for the definition of the closed ones.

The meaning of a closed first-order formula is given as a truth value relative to an interpretation of the types, the predicate constants, and the function constants. To specify the meaning of a non closed formula we also need a valuation of the variables.

Interpretation An interpretation $\mathcal{I}$ of a first-order language $(\mathcal{T}, \mathcal{P}, \mathcal{F})$ is given by

1. a set $[\![\tau]\!]_{\mathcal{I}}$ for each $\tau \in \mathcal{T}$.
2. a function $[\![P]\!]_{\mathcal{I}} : [\![\tau_1]\!]_{\mathcal{I}} \times \cdots \times [\![\tau_n]\!]_{\mathcal{I}} \to \{\mathsf{tt}, \mathsf{ff}\}$ for each $P : [\tau_1, \ldots, \tau_n \to$ boolean$]$.
3. a function $[\![f]\!]_{\mathcal{I}} : [\![\tau_1]\!]_{\mathcal{I}} \times \cdots \times [\![\tau_n]\!]_{\mathcal{I}} \to [\![\tau_{n+1}]\!]_{\mathcal{I}}$ for each function constant $f : [\tau_1, \ldots, \tau_n \to \tau_{n+1}]$.

Such an interpretation associates a truth value with every closed formula and more generally, a function mapping valuations of variables to truth values with every open formula. For pedants we give here a formal definition.

$$
\begin{array}{l}
[\![x]\!]_{\rho,\mathcal{I}} = \rho(x) \\
[\![f(t_1,\ldots,t_n)]\!]_{\rho,\mathcal{I}} = [\![f]\!]_{\mathcal{I}}([\![t_1]\!]_{\rho,\mathcal{I}},\ldots,[\![t_1]\!]_{\rho,\mathcal{I}}) \\
[\![P(t_1,\ldots,t_n)]\!]_{\rho,\mathcal{I}} = [\![P]\!]_{\mathcal{I}}([\![t_1]\!]_{\rho,\mathcal{I}},\ldots,[\![t_1]\!]_{\rho,\mathcal{I}}) \\
[\![\neg\phi]\!]_{\rho\mathcal{I}} = \neg[\![\phi]\!]_{\rho,\mathcal{I}} \\
[\![\phi \star \psi]\!]_{\rho\mathcal{I}} = [\![\phi]\!]_{\rho,\mathcal{I}} \star [\![\phi]\!]_{\rho,\mathcal{I}} \\
[\![\forall x{:}\tau]\!]_{\rho,\mathcal{I}} = \begin{cases} \mathsf{tt}, \text{ if } [\![\phi]\!]_{\rho[x\mapsto v],\mathcal{I}} = \mathsf{tt} \text{ for all } v \in [\![\tau]\!]_{\mathcal{I}} \\ \mathsf{ff}, \text{ otherwise} \end{cases} \\
[\![\exists x{:}\tau]\!]_{\rho,\mathcal{I}} = \begin{cases} \mathsf{tt}, \text{ if } [\![\phi]\!]_{\rho[x\mapsto v],\mathcal{I}} = \mathsf{tt} \text{ for some } v \in [\![\tau]\!]_{\mathcal{I}} \\ \mathsf{ff}, \text{ otherwise} \end{cases}
\end{array}
$$

Here ρ is a partial function on variables.

We note that if ρ is compatible with typing context K in the sense that $\rho(x) \in [\![K(x)]\!]_{\mathcal{I}}$ for all $x \in \mathrm{dom}(K)$, in particular, $\rho(x)$ is defined in this case, then $[\![t]\!]_{\rho,\mathcal{I}} \in [\![\tau]\!]_{\mathcal{I}}$ whenever $K \triangleright_{\mathcal{L}} t : \tau$ and $[\![\phi]\!]_{\rho,\mathcal{I}} \in \{\mathsf{tt},\mathsf{ff}\}$ whenever $K \triangleright_{\mathcal{L}} t$: boolean.

A closed formula is *valid* if its meaning comes out as true under all possible interpretations of the language it is based on. Examples of such valid formulas are as follows.

- $\forall x{:}\tau.P(x) \Rightarrow \exists y{:}\tau.P(y)$ (recall that quantifiers always extend to the left as far as possible),
- $(\forall x{:}\tau_1.R(x, f(x))) \Rightarrow \forall x{:}\tau_1.\exists y{:}\tau_2.R(x, y)$ when $f : [\tau_1 \to \tau_2]$,
- $(\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y)) \Rightarrow (\forall x{:}\tau_1.P(x)) \vee (\forall x{:}\tau_2.Q(x))$,
- $(Q \vee \exists x{:}\tau.P(x)) \Rightarrow \exists x{:}\tau.Q \vee P(x)$ where Q : [$\to$ boolean] is a constant and, moreover, we have a constant $c : [\to \tau]$,
- $\exists x{:}\tau.P(x) \Rightarrow \forall y{:}\tau.P(y)$ again in the presence of a constant $c : [\to \tau]$.

Nullary predicate and function constants are propositional, resp., "ordinary" constants. We may write Q : boolean and $c{:}\tau$ instead of Q : [$\to$ boolean] and $c : [\to \tau]$ to declare them and omit empty parentheses (as done above) when using them.

5.4 First-order sequent calculus

As before we form sequents $\Gamma \Longrightarrow_{\mathcal{L}} \Delta$ from lists of *closed* formulas Γ, Δ over some language $\mathcal{L}$. The meaning of such a sequent is that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ.

Notice that the presence of constants in $\mathcal{L}$ can affect the meaning of a formula hence of a sequent even if these constants do not occur explicitly. This explains the explicit mentioning of $\mathcal{L}$.

We introduce the notation $\mathcal{L}, c{:}\tau$ for the extension of $\mathcal{L}$ with a new constant c of type τ. We keep all the rules for the propositional connectives and add four rules to deal with the quantifiers which we will now explain.

To prove an existential statement we have the rule

$$\frac{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \phi[t/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi} \tag{$\exists$-R}$$

where $\emptyset \triangleright_{\mathcal{L}} t : \tau$.

Here $\phi[t/x]$ denotes the substitution of *closed* term t for variable x in ϕ.

The rule says that to prove an existential statement we must come up with a witness. It corresponds to phrases like "*The desired value x is therefore given by $t\dots$*". Next, we have the following rule to use a universally quantified statement.

$$\frac{\Gamma, \phi[t/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, \forall x{:}\tau.\phi \Longrightarrow_{\mathcal{L}} \Delta} \tag{$\forall$-L}$$

where $\emptyset \triangleright_{\mathcal{L}} t : \tau$.

To use a universally quantified statement we must instantiate it with some concrete term. The rule corresponds to phrases like *"We apply Lemma xxx / the above assumption to $x = t\dots$"* or *"Applying Lemma yyy / the above assumption in this situation yields...*

One should note that these two rules preserve but do not always reflect validity, i.e., it may be that the conclusion of a rule is valid, yet the premise is not. After all, one might have chosen the wrong instantiation. Moreover, it is possible that a universal assumption must be instantiated more than once (consider e.g. an assumption asserting that some relation is transitive), so sometimes one has to keep the quantified formula for later use by prior invocation of rule CONTR.

Next, we have a rule for proving a universal statement:

$$\frac{\Gamma \Longrightarrow_{\mathcal{L}, c:\tau} \Delta, \phi[c/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \forall x{:}\tau.\phi} \tag{$\forall$-R}$$

Here $c : \tau$ is a fresh constant symbol not occurring in $\mathcal{L}$ hence in Γ, Δ, ϕ.

To prove $\forall x{:}\tau.\phi$ we must prove ϕ for a fixed but arbitrary $c : \tau$.

"Fix an arbitrary $c : \tau$... this proves $\forall x{:}\tau.\phi$

Finally, we need a rule to use an existential statement:

$$\frac{\Gamma, \phi[c/x] \Longrightarrow_{\mathcal{L},c:\tau} \Delta}{\Gamma, \exists x{:}\tau.\phi \Longrightarrow_{\mathcal{L}} \Delta} \qquad (\exists\text{-L})$$

To use an existential statement we introduce a fresh name for its witness. We know nothing about the witness except that it satisfies ϕ.

"Lemma xxx provides us with a c such that $\phi[c/x]$"

"Let c be the x provided by (13) above"

We notice that in the latter two rules no formulas with free variables arise as the bound variable is immediately replaced with a fresh constant. There are alternative presentations in which free variables are used for the "fixed but arbitrary constants" occurring in those rules. In a typed setting admitting empty domains of quantification this seems less appropriate as we then would have to annotate each sequent with the set of variables it depends on. Moreover, a variable is supposed to vary, whereas these constants are fixed.

Let's take a look at a couple of representative examples.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(c) \Longrightarrow_{\mathcal{L},c:\tau} P(c)}\ \textsc{Axiom}}{P(c) \Longrightarrow_{\mathcal{L},c:\tau} \exists x{:}\tau.P(x)}\ \exists\text{-R}}{\Longrightarrow_{\mathcal{L},c:\tau} P(c) \Rightarrow \exists x{:}\tau.P(x)}\ \Rightarrow\text{-R}}{\Longrightarrow_{\mathcal{L}} \forall x{:}\tau.P(x) \Rightarrow \exists x{:}\tau.P(x)}\ \forall\text{-R}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(c_1) \vee Q(c_2) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\ \textsc{Prop}}{\forall y{:}\tau_2.P(c_1) \vee Q(y) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\ \forall\text{-L}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\ \forall\text{-L}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L}} \forall x{:}\tau_1.P(x), \forall x{:}\tau_2.Q(x)}\ \forall\text{-R}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L}} (\forall x{:}\tau_1.P(x)) \vee (\forall x{:}\tau_2.Q(x))}\ \vee\text{-R}$$

5.5 Soundness and completeness of first-order sequent calculus

As before we have that a sequent is valid if and only if it is derivable in the sequent calculus, i.e., if there is a proof tree whose leaves are labelled with axioms. Unlike in the propositional case, the contraction rule CONTR is not redundant corresponding to the fact that universal premises may need to be used more than once. This thwarts a naive decision procedure for validity based on constructing a generic proof tree and, indeed, as was shown by TURING validity in first order logic is undecidable. Actually, this result is not very surprising if we consider that basically all of mathematics can be formalised in first-order logic.

Theorem: A sequent is valid if and only if it is derivable in the sequent calculus.

Proof: The "if" direction of the correctness theorem ("soundness") is proved as before by induction on derivations; we simply have to check that all the rules *preserve* validity.

For the "only if" direction ("completeness") we construct a generic proof tree as in the propositional case by breaking down connectives and if nothing else helps instantiating quantifiers $\forall$-L, $\exists$-R. We must make sure that we keep those quantified statements around using CONTR prior to instantiating. If we arrange things in such a way that eventually a quantified formula will be instantiated with every possible term we are sure to find a proof if one exists.

If no proof exists our generic proof tree contains a leaf that is not an axiom or has an infinite path. We can subsume the former (degenerate and rare) case under the latter by applying rule CONTR ad infinitum.

From the infinite path we will construct a counter interpretation by taking terms (also containing the constants newly introduced along the path) to interpret the types, function constants interpreting themselves, and interpreting predicate constants according to how atomic formulas involving them occur along the path. This will ensure that the interpretation falsifies all the sequents along the path hence the root sequent which by assumption has no proof.

Let us look at this in some more detail. Firstly, to counter the information loss in the instantiating rule $\forall$-L and $\exists$-R we replace them by the following combinations with rule CONTR:

$$\frac{\Gamma, \forall x{:}\tau.\phi, \phi[t/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, \forall x.\phi \Longrightarrow_{\mathcal{L}} \Delta} \quad (\forall\text{-L'})$$

$$\frac{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi, \phi[t/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi} \quad (\exists\text{-R'})$$

It is clear that there is a proof with the primed rules iff there is one in the original system.

A *generic proof tree* is a possibly infinite tree labelled with sequents which has the following properties:

1. each internal node is the conclusion of its immediate ancestors by some proof rule.
2. rule WEAK is not used,
3. rule CONTR is used only on atomic non axiom sequents,
4. permutation rules are not used ad infinitum,
5. rules $\forall$-L' and $\exists$-R' are used only with conclusion $\Gamma \Longrightarrow_{\mathcal{L}} \Delta$ where Γ contains atoms and universally quantified formulas only and Δ contains atoms and existentially quantified formulas only. Otherwise we could use one of the validity reflecting rules.
6. no node is labelled with an axiom, i.e., we stop once we have found an axiom
7. on every infinite path starting from $\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\phi$ the formula $\exists x{:}\phi$ is instantiated with all (closed) terms of type τ in $\mathcal{L}$
8. Ditto for infinite paths starting from $\Gamma, \forall x{:}\tau \Longrightarrow_{\mathcal{L}} \Delta$

These properties basically dictate a strategy for obtaining a generic proof tree starting from any sequent. Simply apply the rules backwards with the mentioned restriction on the instantiating rules. When selecting instantiations make sure that every possible instantiation will be eventually chosen unless of course a path ends with an axiom leaf. Please note, that as soon as we make a language extension we must instantiate our quantified formulas with all the terms in the new language as well.

Now suppose that a sequent S has no proof. The generic proof tree constructed from S will then contain an infinite path π (starting from the root). This is "König's Lemma": a finitely branching tree with infinitely many nodes has an infinite path. Along this infinite path π we encounter an increasing (by constants) sequence of languages $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq \ldots$ whose union we call $\mathcal{L}_\infty$. So, a term in $\mathcal{L}_\infty$ will be a term of one of the $\mathcal{L}_i$.

To construct our desired counterinterpretation $\mathcal{I}$ we interpret types by

$$[\![\tau]\!]_{\mathcal{I}} = \{t \mid \emptyset \rhd_{\mathcal{L}_\infty} t : \tau\}$$

We interpret function constants by

$$[\![f]\!]_{\mathcal{I}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

We interpret predicate constants by

$$[\![P]\!]_{\mathcal{I}}(t_1,\ldots,t_n) = \begin{cases} \mathtt{tt}, & \text{if } P(t_1,\ldots,t_n) \text{ is among the antecedents} \\ & \text{(left of } \Longrightarrow\text{) of a sequent in } \pi, \\ \mathtt{ff}, \text{ in all other cases.} \end{cases}$$

Now we show by induction on the size of formulas that whenever a formula ϕ appears as an antecedent of a sequent in π then $[\![\phi]\!]_{\mathcal{I}} = \mathtt{tt}$ and whenever a formula ψ appears as a succedent (to the right of the $\Longrightarrow$) of a sequent in π then $[\![\psi]\!]_{\mathcal{I}} = \mathtt{ff}$, so that in particular all the sequents along π including the root will be falsified by $\mathcal{I}$. So, $\mathcal{I}$ shows that the root is not a valid sequent.

Right, atomic formulas are true under $\mathcal{I}$ precisely if they appear as an antecedent. If an atomic formula appears as a succedent then—since atomic formulas never disappear along the path—it cannot also appear as an antecedent for otherwise we would have an axiom sequent on π contrary to the construction of the generic proof tree. Thus, atomic formulas appearing as succedents are falsified by $\mathcal{I}$. A formula which is not an existentially quantified succedent or a universally quantified antecedent will eventually be broken down by a validity reflecting rule into its subformulas to which the induction hypothesis applies. Consider for example a succedent of the form $\forall x{:}\tau.\phi$. At some point rule $\forall$-L will be applied so $\phi[c/x]$ also occurs as a succedent on π. By induction hypothesis $[\![\phi[c/x]]\!]_{\mathcal{I}} = \mathtt{ff}$, but then $[\![\forall x{:}\tau.\phi]\!]_{\mathcal{I}} = \mathtt{ff}$, too.

If, finally, we have an existentially quantified succedent, e.g., $\exists x{:}\phi$ then by the "round robin" policy used for instantiating all formulas $\phi[t/x]$ with $\emptyset \rhd_{\mathcal{L}_\infty} t : \tau$ will occur as succedents along π hence are falsified by $\mathcal{I}$. Since $[\![\tau]\!]_{\mathcal{I}}$ comprises precisely all those terms we conclude that $[\![\exists x{:}\tau.\phi]\!] = \mathtt{ff}$, as well. The case of a universally quantified antecedent is analogous.

One should not underestimate the power of first-order logic. Even without function constants the counter interpretation may be infinite due to infinitely many newly introduced constants. Consider for example the formula

$$(\forall x,y,z{:}\tau.R(x,y) \wedge R(y,z) \Rightarrow R(x,z)) \wedge (\forall x{:}\tau.\exists y{:}\tau.R(x,y)) \Rightarrow \exists x{:}\tau.R(x,x)$$

It is not valid but holds in all finite interpretations. You may find it instructive to form the generic proof tree for this formula.

5.6 First-order logic in PVS

Language concepts are declared anywhere in a theory, but before being used

```
D, T1, T2 : TYPE+
c : T1
P : [D -> boolean]
Q : boolean
f : [T1->T2]
```

Here `TYPE+` stands for *nonempty* type. There is also the declaration `T:TYPE` which stands for a possibly empty type. In this case it would not be allowed to declare a constant of type `T`.

This design decision of PVS is open to debate. By declaring a constant of a type we explicitly state that it is nonempty so why say it twice?

Quantifiers are written `FORALL(x:T):` and `EXISTS(x:T):` Do not forget the colon after the parenthesis.

```
exI : THEOREM
    FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)
orex : THEOREM
     (Q OR EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q OR P(x)
depp : THEOREM
      EXISTS(x:D): FORALL(y:D): P(x) IMPLIES P(y)
gen : THEOREM
      (EXISTS(x:D):P(x)) IMPLIES FORALL (x:D):P(x)
```

The rules $\forall$-L and $\exists$-R are invoked with the command `inst` (instantiation). The rules $\forall$-L and $\exists$-R are invoked with the command `skolem` (after TH. SKOLEM).

The `inst` command takes as argument a formula number (the formula to be instantiated) and a term to instantiate with. For example, in the situation

```
{-1} P(d)
  |-------
{1}  EXISTS (x:T1): P(x)
```

the command

```
(inst 1 "d")
```

leads to

```
{-1} P(d)
  |-------
{1}  P(d)
```

which is an axiom.

The `skolem` command takes as argument a formula number and the name of a new constant. If it isn't fresh then PVS complains. For example, in the situation

```
  |-------
{1}    FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)
```

The command `(skolem 1 "c")` is no good because we have already used `c` for a constant above. However, `(skolem 1 "d")` succeeds and gives

```
  |-------
{1}    P(d) IMPLIES EXISTS(x:D): P(x)
```

The command `M-x show-skolem-constants` displays all the constants introduced in the course of the proof.

The commands `inst` and `skolem` allow the treatment of several variables at once. There are also the derived form `inst?` which guesses an appropriate instantiation heuristically (alas often quite badly) and `skolem!` which automatically introduces as many constants as possible (making up fresh names for them). Furthermore, `skosimp` is a combination of `skolem!` and simplification. See the PVS prover guide for details.

As an exercise try to prove all of the "theorems" below.

```
fol: THEORY
 BEGIN
 D, T1, T2 : TYPE+
 c : T1
 d : D
 P : [D -> boolean]
 Q : boolean

  allE : THEOREM
  FORALL(x:D): (FORALL(y:D): P(y)) IMPLIES P(x)



 andall : THEOREM
   (Q AND (FORALL(x:D):P(x))) IMPLIES FORALL(x:D):Q AND P(x)

 exI : THEOREM
```

```
   FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)

 andex : THEOREM
   (Q AND EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q AND P(x)

 orex : THEOREM
     (Q OR EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q OR P(x)

 depp : THEOREM
      EXISTS(x:D): FORALL(y:D): P(x) IMPLIES P(y)

 doub : THEOREM
      FORALL(x,y:D) : EXISTS (z:D) : P(z) IMPLIES P(x) & P(y)

 P1 : [T1->boolean]
 P2 : [T2 -> boolean]

 por : THEOREM
     (FORALL(x:T1,y:T2):P1(x) OR P2(y))  IMPLIES
                   (FORALL(x:T1):P1(x)) OR (FORALL(x:T2):P2(x))

 R : [D,D->boolean]

 per: THEOREM
       (FORALL (x,y:D):R(x, y) IMPLIES R(y, x)) AND
       (FORALL (x,y,z:D): R(x, y) AND R(y, z) IMPLIES R(x,z)) IMPLIE
       (FORALL (x:D): (EXISTS (y:D): R(x,y)) IMPLIES R(x,x))
 END fol
```

6 First-order resolution

As in the propositional case the method of resolution provides a generally more efficient way to decide validity of formulas than proof search in Gentzen's sequent calculus. In the first-order case we may instantiate universally quantified variables prior to resolving so as to achieve agreement of literals. For example, we may resolve the clauses $\{P(f(x, g(y))), Q(x)\}$ which denotes $\forall x, y.P(f(x, g(y))) \vee Q(x)$ (types omitted) and $\{\neg P(f(g(z), w)), R(w)\}$ which denotes $\forall w, z.\neg P(f(g(z), w)) \vee$

$$
\begin{array}{lll}
C_1 = \{P(f(x, g(y))), Q(x)\} & \text{, i.e.,} & \forall x, y.P(f(x, g(y))) \vee Q(x) \\
C_2 = \{\neg P(f(g(z), w)) \vee R(w)\} & \text{, i.e.,} & \forall w, z.\neg P(f(g(z), w)) \vee R(w)\} \\
 & \text{resolve to} & \\
C_3 = \{R(g(y)) \vee Q(g(z))\} & \text{, i.e.,} & \forall y, z.R(g(y)) \vee Q(g(z))
\end{array}
$$

Figure 20: Example of resolution

$R(w)$ to form $\{R(g(y)), Q(g(z))\}$ which denotes $\forall y, z.R(g(y)) \vee Q(g(z))$.

Notice that again a satisfying interpretation for the former two clauses will also satisfy the latter. In this example, we could also have instantiated x with something like $g(f(h(c())))$ and accordingly y with $f(h(c()))$. However, in order to maximise future success it is advisable to choose the instantiation which makes the least possible commitment or, in formal terms, the *most general unifier*. While this is in practice always done it is, for the purpose of establishing completeness, easier to allow arbitrary instantiations.

Let us look at the details. First-order resolution operates on clauses which are sets of first-order literals, i.e., negated or non-negated atomic formulas, which are understood as being universally quantified over the variables they contain. In order to avoid problems with empty types we assume that our language is such that every type contains at least one closed term.

- A first-order literal is a negated or non-negated atomic formula.
- A first-order clause is a set of first-order literals
- It denotes the disjunction of the literals universally quantified over the variables

Given a set of first-order clauses $\mathcal{C}$ we can use first-order resolution to decide whether it is satisfiable, i.e., whether there exists an interpretation which makes it true. If we can derive the empty clause from $\mathcal{C}$ by successive application of rules INST and RES then surely $\mathcal{C}$ is unsatisfiable. Conversely, if $\mathcal{C}$ is unsatisfiable then it is possible to derive the empty clause. The proof of this result is based on correctness of propositional resolution and Herbrand's theorem which asserts that a set of formulas of the form $\forall \vec{x}{:}\vec{\tau}.\phi$ with ϕ quantifier-free is satisfiable if and only if the set of its closed instantiations is propositionally satisfiable:

Theorem ("Herbrand's theorem"): Let $\mathcal{S}$ be a set of formulas of the form $\forall \vec{x}.\phi$ with ϕ quantifier free.

- Instantiation rule:

$$\frac{C}{C[t_1/x_1,\ldots,t_n/x_n]} \qquad \text{(INST)}$$

  where x_i are the variables mentioned in C and the t_i are *possibly open* terms (of the right type!).

- Resolution rule

$$\frac{C_1 \cup \{A\} \qquad C_2 \cup \{\neg A\}}{C_1 \cup C_2} \qquad \text{(RES)}$$

- Side condition: there is a closed term of each type.

- Aim: try to derive empty clause from initial set so as to show unsatisfiability.

Figure 21: First-order resolution

Define

$$\overline{\mathcal{S}} := \{\phi[t_1/x_1,\ldots,t_n/x_n)] \mid \forall x_1{:}\tau_1,\ldots\forall x_n{:}\tau_n.\phi \in \mathcal{S} \text{ and } \emptyset \rhd_{\mathcal{L}} t_i{:}\tau_i\}$$

as the set of closed instantiations of formulas in $\mathcal{S}$.

There exists an interpretation $\mathcal{I}$ validating all formulas in $\mathcal{S}$ iff there exists a propositional valuation η of the atomic formulas (viewed as propositional atoms) validating all formulas in $\overline{\mathcal{S}}$.

Proof: Given $\mathcal{I}$ define η by $\eta(P(t_1,\ldots,t_n)) = [\![P(t_1,\ldots,t_n)]\!]_{\mathcal{I}}$. Given η interpret types as sets of (closed) terms, function symbols by themselves, and predicates as given by η. $\Box$

Thus to establish unsatisfiability of a set of first-order clauses it is enough to establish propositional unsatisfiability of their closed instantiations, but that's precisely what rule RES can do as shown in Section 4.

Rule INST on the other hand, allows us to generate the set of closed instantiations. Performing resolution on clauses containing (universally quantified) variables certainly does no harm, but may of course speed up success.

6.1 Most general unifiers

As already mentioned, in practice one resolves clauses by instantiating with the *most general unifier*. The most general unifier of two open terms $u(x_1,\ldots,x_m)$

- $u(x_1, \ldots, x_m), v(y_1, \ldots, y_n)$ two term with free variables $\vec{x}$ and $\vec{y}$.
- Most general unifier consists of $\vec{s}(\vec{z})$ and $\vec{t}(\vec{z})$ such that $u(\vec{s}(\vec{z})) = v(\vec{t}(\vec{(z)}))$ and
- whenever $u(\vec{s'}) = v(\vec{t'})$ then $s' = \vec{s}(\vec{a}), t' = \vec{t}(\vec{b})$.
- **Example:** $u = f(x, g(y)), v = f(h(y), x)$:
  $\vec{s} = [h(z)/x, x/y], \vec{t} = [z/y, g(x)/x]$
- **Note:** the most general unifier might not exist, e.g., $s = f(x), t = g(y)$.

Figure 22: Most general unifier

and $v(y_1, \ldots, y_n)$ consists of two sequences of open terms $t_1, \ldots, t_m$ and $s_1, \ldots, s_n$ involving variables $z_1, \ldots, z_k$, such that $u(t_1, \ldots, t_m) = v(s_1, \ldots, s_n)$ and, moreover, any instantiation making u equal to v arises from this one by instantiation, i.e., whenever $u(\vec{t'}) = v(\vec{s'})$ then $\vec{t'} = \vec{t}[\vec{a}/\vec{z}]$ and $\vec{s'} = \vec{s}[\vec{b}/\vec{z}]$ for some $\vec{a}, \vec{b}$.

For example, the most general unifier of $f(x, g(y))$ and $f(h(y), x)$ is $\vec{s} = [h(z)/x, x/y], \vec{t} = [z/y, g(x)/x]$ because we have $f(x, g(y))[h(z)/x, x/y] = f(h(z), g(x)) = f(h(y), x)[z/y, g(x)/x]$. Notice that the variable names in u, v as well as the common ones in the s, t are rather arbitrary. In particular, if the same variable happens to occur in both u and v, we can instantiate it differently in both.

The most general unifier is effectively found by comparing the terms in question in a top down fashion starting from the outermost function constant. We omit the details of the unification algorithm and also a formal proof that resolution with most general unifiers is complete. The idea is to map any proof using RES and INST to a proof using only the following combined rule.

$$\frac{C_1 \cup \{A_1\} \qquad C_2 \cup \{\neg A_2\} \qquad \vec{s}, \vec{t}\text{ m.g.u. of } A_1, A_2}{C_1[\vec{s}] \cup C_2[\vec{t}]} \ (\text{RES-UNIF})$$

Of course it goes without saying that all the instantiations made in the course of resolution must be type correct, i.e., the resulting terms and atomic formulas must be well-formed.

At this point it is worth reiterating the point made earlier in Section 5 about types separating interesting and potentially difficult facts from uninteresting obvious facts. The number of clauses hence the search space for resolution becomes smaller the more we make use of types.

$$
\begin{array}{ll}
\{\ \{\neg S(r), \neg A(r,x,y), A(r,y,x)\}, & (\mathbf{C1}) \\
\quad \{A(r,f(r),g(r)), S(r)\}, & (\mathbf{C2}) \\
\quad \{\neg A(r,g(r),f(r)), S(r)\}, & (\mathbf{C3}) \\
\quad \{S(s())\}, & (\mathbf{C4}) \\
\quad \{A(s(),a(),b())\}, & (\mathbf{C5}) \\
\quad \{\neg A(s(),b(),a())\}\ \} & (\mathbf{C6})
\end{array}
$$

Figure 23: Example of first-order resolution

6.2 Skolemisation

We now discuss how to translate arbitrary first-order formulas into first-order clauses; somewhat surprisingly, any first-order formula is equivalent to set of first-order clauses albeit in a richer language.

In order to do that we use the following fact known as *skolemisation*, again after TH. SKOLEM.

Fact: Let $\psi := \forall x_1{:}\tau_1 \ldots \forall x_n{:}\tau_n.\exists y{:}\tau_{n+1}.\phi(\vec{x}, y)$ in some language $\mathcal{L}$ and let $\mathcal{L}'$ be the language $\mathcal{L}$ extended with a new function constant $f : [\tau_1, \ldots, \tau_n \to \tau_{n+1}]$. The formula ψ is satisfiable if and only if the formula $\forall x_1{:}\tau_1 \ldots \forall x_n{:}\tau_n.\phi(\vec{x}, f(\vec{x}))$ is satisfiable.

The function constant f is called a *Skolem function*.

Now consider an arbitrary first-order formula ϕ. Using the following tautologies

$$
\begin{array}{ll}
\alpha \star (Qx{:}\tau.\beta(x)) \Leftrightarrow Qx{:}\tau.\alpha \vee \beta(x) & \star \in \{\vee, \wedge, \Rightarrow\}, Q \in \{\forall, \exists\} \\
(Qx{:}\tau.\alpha(x)) \Rightarrow \beta \Leftrightarrow \bar{Q}x{:}\tau.\alpha(x) \Rightarrow \beta & \\
(\neg Qx{:}\tau.\alpha(x)) \Leftrightarrow \bar{Q}x{:}\tau.\neg\alpha(x) &
\end{array}
$$

we can bring each formula (up to equivalence) into the form

$$Q_1 x_1{:}\tau_1.Q_2 x_2{:}\tau_2.\ldots.Q_n x_n{:}\tau_n.\phi_0$$

with $Q_1, \ldots, Q_n \in \{\forall, \exists\}$ and ϕ_0 quantifier-free. The latter formula is by definition in *prenex form*.

Thereafter, using the "fact" we can successively replace existential quantifiers with new function constants that take all the previous universally quantified variables as arguments so as to obtain a universally quantified boolean combination of atomic formulas which in turn is equivalent to a set of first-order clauses. Summing up, we have the following result.

Theorem: For every first-order formula ϕ one can effectively find a set of first-order clauses $\mathcal{C}$ such that ϕ is satisfiable iff $\mathcal{C}$ is.

Proof: Bring ϕ into prenex form (move all quantifiers to the front exchanging $\forall$ and $\exists$ when moving out of a negative position). Replace existential quantifiers by *Skolem functions* using language extension by function constants. Bring the resulting universally quantified formula into clausal form as in propositional case. $\square$

In my view, the reason why resolution is superior to proof search in sequent calculus is that the choice of instantiations is made after looking at two clauses (and performing unification) which, when informally translated back to sequent calculus, means that the form of the side formulas Γ, Δ in a sequent, say, $\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi$ help in finding an appropriate instantiation for x. I am not aware of a precisation of this argument in the form of a unification-based strategy for finding instantiations in sequent calculus proof search. In this context one should note that the flattening of nested quantifications using Skolemisation is crucial for the success of unification.

6.3 Some puzzles

Here are some small examples that should be brought into clausal form and proved by hand, using PVS, or automatically using SPASS.

The mislabelled boxes (from `http://www.cs.miami.edu/~tptp/`): There are three boxes a, b, and c on a table. Each box contains apples or bananas or oranges. No two boxes contain the same thing. Each box has a label that says it contains apples or says it contains bananas or says it contains oranges. No box contains what it says on its label. The label on box a says "apples". The label on box b says "oranges". The label on box c says "bananas". You pick up box b and it contains apples. What do the other two boxes contain?

Barber's problem (from `http://www.cs.miami.edu/~tptp/`): There is a barbers' club that obeys the following three conditions:

1. If any member has shaved any other member – whether himself or another – then all members have shaved him, though not necessarily at the same time.
2. Four of the members are named Guido, Lorenzo, Petrucio, and Cesare.
3. Guido has shaved Cesare. Prove Petrucio has shaved Lorenzo