UNITPROP$(\mathcal{C}, \eta) =$
IF $\mathcal{C}[\eta]$ contains the empty clause
let $\rho \subseteq \eta$ be the smallest sub-valuation of η such that $\emptyset \in \mathcal{C}[\rho]$
RETURN $(\rho,$ "unsatisfiable")
ELSE IF $\mathcal{C}[\eta]$ is empty
RETURN $(\eta,$ "satisfiable")
ELSE IF $\mathcal{C}[\eta]$ contains a unit clause $\{A\}$
RETURN *UNITPROP*$(\mathcal{C}, \eta[A \mapsto \mathsf{tt}])$
ELSE IF $\mathcal{C}[\eta]$ contains a unit clause $\{\neg A\}$
RETURN *UNITPROP*$(\mathcal{C}, \eta[A \mapsto \mathsf{ff}])$
ELSE RETURN $(\eta,$ "undecided")

DPLL$(\eta) =$
$(\eta', v) :=$ *UNITPROP*$(\mathcal{C}, \eta)$
IF $v =$ "satisfiable"
RETURN "satisfiable"
ELSE IF $v =$ "unsatisfiable"
$\mathcal{C} := \mathcal{C} \cup \{k\}$where k asserts that at least one atom is valued different from η'.
RETURN "unsatisfiable"
ELSE
choose an atom $A \notin \mathrm{dom}(\eta')$
IF *DPLL*$(\eta'[A \mapsto \mathsf{tt}]) =$"satisfiable"
RETURN "satisfiable".
ELSE
RETURN *DPLL*$(\eta'[A \mapsto \mathsf{ff}])$

INPUT a set of clauses $\mathcal{X}$
$\mathcal{C} := \mathcal{X}$
OUTPUT *DPLL*$(\emptyset)$

Figure 16: DPLL-algorithm with clause learning

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 17: First-order formulas (informal)

restrict the range of quantifiers. Figure 18 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 d and 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".

1. $\forall x.\text{student}(x) \Rightarrow \exists n.\text{number}(x) \wedge \text{has\_matric\_no}(x, n)$

2. $\forall x.\text{student}(x) \Rightarrow \neg\text{has\_matriculated}(x) \Rightarrow \text{will\_be\_expelled}(x)$

3. $\forall x.\text{human}(x) \Rightarrow \text{philosopher}(x)$

4. $\exists x.\text{student}(x) \wedge \forall c.\text{course}(c) \Rightarrow \text{complains\_about}(x, c)$

5. $\exists x.\forall y.\neg(y \in x)$

6. $\forall n.\text{number}(n) \Rightarrow \exists d.\text{number}(d) \wedge \text{leq}(\text{power}(2, d), n) \wedge \text{lt}(n, \text{power}(2, \text{plus}(d, 1)))$

7. $\forall x.\text{bilg}(x) \Rightarrow \text{prugl}(\text{knirp}(x)) \wedge \forall y.\text{quist}(y) \Rightarrow \neg\text{bebelfs}(\text{knirp}(x), y)$

Figure 18: Formalisation in untyped first-order logic

1. $\forall x{:}\text{student}.\exists n{:}\text{number}.\text{has\_matric\_no}(x, n)$

2. $\forall x{:}\text{student} \Rightarrow \neg\text{has\_matriculated}(x) \Rightarrow \text{will\_be\_expelled}(x)$

3. $\forall x{:}\text{human}.\text{philosopher}(x)$

4. $\exists x{:}\text{student}.\forall c{:}\text{course}.\text{complains\_about}(x, c)$

5. $\exists x{:}\text{set}.\forall y{:}\text{set}.\neg(y \in x)$

6. $\forall n{:}\text{number}.\exists d{:}\text{number}.\text{leq}(\text{power}(2, d), n)\wedge$
   $\text{lt}(\text{power}(2, \text{plus}(d, 1)))$

7. $\forall x{:}\text{bilg}.\text{prugl}(\text{knirp}(x)) \wedge \forall y{:}\text{quist}.\neg\text{bebelfs}(\text{knirp}(x), y)$

Figure 19: Typed first-order logic

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* $[\tau_1, \ldots, \tau_n \rightarrow \text{boolean}]$ where $\tau_1, \ldots, \tau_n \in \mathcal{T}$
3. a collection $\mathcal{F}$ of function constants each endowed with an arity $[\tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}]$ 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 $P : [\tau_1, \ldots, \tau_n \rightarrow \text{boolean}]$ and $f : [\tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}]$ to indicate the arities of predicate, resp. function constants.

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

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

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

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

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

$$\frac{K(x) = \tau}{K \rhd_{\mathcal{L}} x : \tau} \quad (\textsc{Var})$$

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

Figure 20: Well formed terms

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 $K \rhd_{\mathcal{L}} t : \tau$ 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 $K \rhd_{\mathcal{L}} \phi :$ 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 $P : [\tau_1, \ldots, \tau_n \to \text{boolean}]$.
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.

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

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 $[\![\phi]\!]_{\rho,\mathcal{I}} \in \{\mathtt{tt}, \mathtt{ff}\}$ 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 $Q : [\to \text{boolean}]$ 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 $Q : [\to \text{boolean}]$ 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…*". 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 $x = t$… "* 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:

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

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.

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:

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

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

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

We also note that the new rules preserve and reflect validity: the conclusion of any one of the new rules is valid iff its premise is.

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. 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.

4. no internal node is labelled with an axiom, i.e., we stop once we have found an axiom

5. 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 $\mathcal{L}$

6. 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 rules that instantiate quantifiers. 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 might have a finite path ending in a non axiom consisting of atomic formulas only. In this case, we can argue as in the case of propositional logic that the root sequent is not valid.

Alternatively, and this is the interesting case, the generic proof tree will 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

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

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 $[\![\phi]\!]_{\mathcal{I}} = \mathsf{tt}$ and whenever a formula ψ appears as a succedent (to the right of the $\Longrightarrow$) of a sequent in π then $[\![\psi]\!]_{\mathcal{I}} = \mathsf{ff}$, 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.

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 the induction hypothesis $[\![\phi[c/x]]\!]_{\mathcal{I}} = \mathsf{ff}$, but then $[\![\forall x{:}\tau.\phi]\!]_{\mathcal{I}} = \mathsf{ff}$, 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, all the 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 $[\![\exists x{:}\tau.\phi]\!] = \mathsf{ff}$, as well. The case of a universally quantified antecedent is analogous. This completes the proof.

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 ∀-L and ∃-R are invoked with the command `inst` (instantiation). The rules ∀-L and ∃-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 $\forall w, z.\neg P(f(g(z),w)) \vee R(w)$ 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