Robert Constable and Christoph Kreitz

Logics

Truth through evidence

Springer

Chapter 8
First-Order Logic

8.1 Overview

First-Order Logic is the calculus one usually has in mind when using the word "logic". It is widely used in mathematics and computer science because it is expressive enough for all of mathematics, except for those concepts that rely on a notion of construction or computation. However, dealing with more advanced concepts is often somewhat awkward and researchers often design specialized logics for that reason.

In our account of propositional logic we already made use of first-order logic to describe metamathematical concepts such as tautologies, soundness, or completeness. For instance, the definition of a formula X being a tautology, which requires the boolean value of X under an arbitrary assignment of values to the variables of X to be true, is expressed as $\forall v{:}\mathrm{Var}(X)\to\mathbb{B}.\ (\mathrm{bval}(X,v)=t$, and the goal of tableaux proofs (find a falsifying assignment) as $\exists v{:}\mathrm{Var}(X)\to\mathbb{B}.\ (\mathrm{bval}(X,v)=f$. Using formulas to represent metamathematical concepts makes the descriptions more precise and provides a foundation for the implementation of logical calculi on a computer.

Our account of first-order logic will be similar to the one of propositional logic. We will present

- The *syntax*, or the formal language of first-order logic, that is symbols, formulas, sub-formulas, formation trees, substitution, etc.
- The *semantics* of first-order logic
- *Proof systems* for first-order logic, such as the axioms, rules, and proof strategies of the first-order tableau method and refinement logic
- The *meta-mathematics* of first-order logic, which established the relation between the semantics and a proof system

In many ways, first-order logic is a straightforward extension of propositional logic. One must, however, be aware that there are subtle differences.

8.2 Syntax

The syntax of first-order logic is essentially an extension of propositional logic by quantification $\forall$ and $\exists$. Propositional variables are replaced by *n-ary predicate symbols* (P, Q, R) which may be instantiated with either *variables* (x, y, z, ...) or *parameters* (a, b, ...). Here is a summary of the most important concepts.

Definition 8.1 (Syntax of first-order logic).

(1) Atomic formulas are expressions of the form $Pc_1..c_n$ where P is an n-ary predicate symbol and the c_i are variables or parameters.
(2) The formulas of first-order logic are recursively defined as follows
  a. Every atomic formula is a formula.
  b. If A is a formula then so is $\neg A$.
  c. If A and B are formulas then so are $(A \Rightarrow B)$, $(A \wedge B)$, and $(A \vee B)$.
  d. If B is a formula and x a variable then $(\forall x)B$ and $(\exists x)B$ are formulas.
(3) Pure formulas are formulas without parameters.
(4) The scope of a quantifier is the smallest formula that follows the quantifier.
(5) The degree $d(A)$ of a formula A is the number of logical connectives and quantifiers in A.

Note that many accounts of first-order logic use *terms* built from variables and *function symbols* instead of parameters. This makes the formal details a bit more complex but the fundamental concepts remain the same.

Note that conventions about the scope of quantifiers differ in the literature and that many accounts of logic provide preference rules that permit dropping parentheses. As this usually leads to confusion we require that parentheses should always be used to avoid ambiguities. Outer parentheses and parentheses around quantified formulas may be omitted.

Thus in $(\forall x)Px \vee Qx$ the scope of $(\forall x)$ is just Px, while Qx is outside the scope of the quantifier. To include Qx in the scope of $(\forall x)$ one has to add parentheses: $(\forall x)(Px \vee Qx)$.

Definition 8.2 (Free and bound variables / Substitution).

(1) A variable x occurs *bound* in A if it occurs in the scope of a quantifier $(\forall x)$ or $(\exists x)$. Any other occurrence of x in A is *free*.
(2) Closed formulas (or sentences) are formulas without free variables.
(3) Substitution: $A|_a^x$ (or $A[a/x]$) is the result of replacing every free occurrence of the variable x in A by the parameter a.[1]
(4) Subformulas are defined similar to propositional logic. The only modification is that for any parameter a the formula $B|_a^x$ is an immediate subformula of $(\forall x)B$ and $(\exists x)B$.
(5) The formation tree of a formula F is a representation of all subformulas of A in tree format.
  a. The root of the tree is F.

[1] Since we only allow variables to be substituted by terms, issues like capture cannot occur

b. The sucessor of a formula of the form $\neg A$ is A.
c. The successors of $A \wedge B$, $A \vee B$, $A \Rightarrow B$ are A and B.
d. The successors of $(\forall x)B$ and $(\exists x)B$ are $A|_{a_i}^{x}$ for all parameters a_i.
Note that quantifiers usually have infinitely many successors.
e. Atomic formulas have no successors.

From now on, closed formulas are the default when we use the word "formula".

8.3 Evidence Semantics

As in the propositional case, the meaning of formulas will be defined through evidence. A formula X is considered *valid* if we are able to provide evidence that justifies the validity of X. Evidence will described in the form of terms that can be used in a computational fashion. In a sense, evidence terms form a primitive functional programming language that can be executed on a computer.

- Evidence for atomic formulas
A predicate symbol P is a placeholder for some arbitrary unknown predicate and an atomic formula $A = Pc_1..c_n$ is a placeholder for some unknown proposition. As such, it cannot have a fixed evidence and the type of its evidences $[A]$ remains unspecified.[2]
- The evidence terms for $A \Rightarrow B$, $A \wedge B$, $A \vee B$, $\neg A$ are formal justifications for implications, conjunctions, disjunctions, and negations as in the case of propositional logic. They are summarized in the upper part of table 8.1.
- Evidence for $(\forall x)B$
To know $(\forall x)B$ we must know how to generate some evidence b for B for every possible instance of the variable x. Thus the evidence for $(\forall x)B$ must be a *function* f that generates the evidence $b : [B]$ for every possible value a that we may provide as input.
To some extent the evidence for $(\forall x)B$ is similar to the one for $A \Rightarrow B$ as in both cases the type of evidences is a function type. There are, however, some important differences. While in the case of implication the input type is some evidence type $[A]$ the input for evidences for $(\forall x)B$ is taken from some (unspecified) *universe* of objects, which we denote by $\mathbb{U}$. Furthermore, since the formula B may contain x as free variable, there is some dependency between the *value of the input* $a : \mathbb{U}$ and the *type of the output* of the evidence function.
Consider, for instance, the formula $(\forall x)(Pa \Rightarrow Px)$. Here the subformula $B = (Pa \Rightarrow Px)$ has evidence if we instantiate x with a and no evidence otherwise. So the evidence type for B clearly depends on the object we choose as input for the evidence function $f : [(\forall x)B]$.

[2] This may change when we study advanced logics in which certain predicate symbols like the equality symbol have a predefined meaning. Atomic formulas like `0=0` will then have atomic evidences.

As a consequence, the evidence type of $(\forall x)B$ cannot be a simple function type like $\mathbb{U}\to[B]$ but has to be a function type that captures the dependency between input value and output type. In programming, such types are called *dependent function types*.
Adopting programming notation we write $[(\forall x)B] = x:\mathbb{U}\to[B]$ [3]
Let us look at two simple concrete examples.

- $(\forall x)(Px \Rightarrow Px)$: The evidence must be a function $f : (x{:}\mathbb{U}\to([Px]\to[Px]))$ that takes some object x and produces evidence for $Px \Rightarrow Px$. The latter is a function that takes some evidence $p : [Px]$ and produces evidence for Px. Using identity function $\lambda p.\, p$ as solution for that the overall evidence has the form $\lambda x.\,(\lambda p.\, p)$.
- $((\forall x)Px) \Rightarrow Pa$: The evidence must be a function that takes as input some evidence $f : [(\forall x)Px]$ and produces evidence for Pa. Since f has the type $x : \mathbb{U}\to[Px]$, applying f to the parameter a will result in an element of $[Pa]$, which we can use as the desired evidence. Thus the overall evidence will have the form $\lambda f.\, fa$.

- Evidence for $(\exists x)B$
  To know $(\exists x)B$ we must be able to generate evidence for B for at least one possible instance of the variable x. As in the case of the universal quantifier the formula B may contain x as free variable, which means that there is some dependency between element that instantiates x and the type of the evidence for B that we need to provide. Therefore we cannot simply provide some element of $[B]$ as evidence for $(\exists x)B$ but we also have to provide the specific element $a : \mathbb{U}$ for which we claim B to be valid.
  Therefore the evidence for $(\exists x)B$ must give us the concrete witness a *and* the evidence $b : B[a/x]$, that is a pair (a,b) where $a : \mathbb{U}$ and $b : [B[a/x]]$. Thus the type of evidences $[(\exists x)B]$ is a *dependent product type*.
  We write $[(\exists x)B] = x : \mathbb{U}\times[B]$.
  Again let us look at a few concrete examples
  - $Pa \Rightarrow ((\exists x)Px)$: The evidence must be a function that takes some evidence $p : [Pa]$ and produces a pair (x,p') where $x : \mathbb{U}$ and p' is evidence for Px. The obvious choice is $x = a$ and $p' = p$ and the the overall evidence is $\lambda p.\,(a,p)$.
  - $((\exists x)Px) \Rightarrow ((\exists y)Py)$: The evidence must be a function that takes as input an element $z : (x : \mathbb{U}\times[Px])$, i.e. a pair $z\ (a,p)$, where $a : \mathbb{U}$ and p is evidence for Pa, and produces a pair (x,p') where $x : \mathbb{U}$ and p' is evidence for Px. The obvious choice is again $x = a$ and $p' = p$ and the the overall evidence is $\lambda z.\,(z_1,z_2)$. Since (z_1,z_2) is identical to z we can simplify the evidence to $\lambda z.\,z$. [4]

---

[3] Dependent function types are often written as $x : S\to T[x]$, where $T[x]$ expresses that the output type T may contain x as free variable. $x : S\to T[x]$ is the the type of functions that on input $s : S$ return a value of a type $T[s/x]$.

[4] One could also attempt to argue semantically, claiming that $(\exists x)Px$ and $(\exists y)Py$ must express the same, and provide $\lambda x.x$ as overall evidence. But the justification for this claim is the argument that we just provided.

Table 8.1 summarizes the evidence terms and types for the first-order quantifiers. Note that the evidence terms associated with universal quantifiers are the same as those associated with implication and that he evidence terms associated with existential quantifiers are the same as those associated with conjunction. The difference is only in the evidence types – quantifiers require dependent types while "ordinary"function and product types are sufficient for the propositional connectives.

proposition A	evidence type $[A]$	evidence term	evidence decomposition
$A \Rightarrow B$	$[A] \to [B]$	$\lambda a.b$	$f(a)$
$A \wedge B$	$[A] \times [B]$	(a,b)	p_1, p_2
$A \vee B$	$[A] + [B]$	$\texttt{inl}(a), \texttt{inr}(b)$	$\texttt{case } e \texttt{ of inl}(a) \to s \mid \texttt{inr}(b) \to t$
$\neg A$	$[A] \to \{\}$	$\lambda a.b$	$f(a)$
$(\forall x)B$	$x : \mathbb{U} \to [B]$	$\lambda a.b$	$f(a)$
$(\exists x)B$	$x : \mathbb{U} \times [B]$	(a,b)	p_1, p_2

Table 8.1 Evidence terms for first-order logic

We conclude this section by posing a few problems that should be elaborated in groups. For each of the formulas below the group should find the evidence that validates the formula or explain why evidence cannot be constructed.

(1) $((\forall x)(Px \wedge Qx)) \Rightarrow ((\forall x)Px \wedge (\forall x)Qx)$:
The evidence must be a function that takes as input a function $f : (x{:}\mathbb{U} \to [Px] \times [Qx])$ and produces a pair of functions $(g,h) : (x{:}\mathbb{U} \to [Px]) \times (x{:}\mathbb{U} \to [Qx])$. To construct g and g we take an input $x{:}\mathbb{U}$ and generate the evidence for $[Px]$ by applying f to x and selecting the first and second component. Thus the overall evidence is $\lambda f.(\lambda x.(f\,x)_1, \lambda x.(f\,x)_2)$

(2) $((\forall x)Px \wedge (\forall x)Qx) \Rightarrow ((\forall x)(Px \wedge Qx))$: The evidence is $\lambda z.(\lambda x.(z_1\,x, z_2\,x))$

(3) $((\forall x)Px \vee (\forall x)Qx) \Rightarrow ((\forall x)(Px \vee Qx))$:
The evidence is $\lambda z.(\lambda x.(\texttt{case } z \texttt{ of inl}(f) \to \texttt{inl}(f\,x) \mid \texttt{inr}(g) \to \texttt{inr}(g\,x)))$

(4) $((\forall x)(Px \vee Qx)) \Rightarrow ((\forall x)Px \vee (\forall x)Qx)$: This formula is not valid.
A possible counterexample is Px being $\texttt{odd}(x)$ and Qx being $\texttt{even}(x)$.

(5) $((\exists x)(Px \wedge Qx)) \Rightarrow ((\exists x)Px \wedge (\exists x)Qx)$: The evidence is $\lambda z.((z_1, z_{21}), (z_1, z_{22}))$

(6) $((\exists x)Px \wedge (\exists x)Qx) \Rightarrow ((\exists x)(Px \wedge Qx))$: This formula is not valid.
A possible counterexample is Px being $\texttt{odd}(x)$ and Qx being $\texttt{even}(x)$.

(7) $((\exists x)Px \vee (\exists x)Qx) \Rightarrow ((\exists x)(Px \vee Qx))$:
The evidence is $\lambda z.(\texttt{case } z \texttt{ of inl}(x) \to (x_1, \texttt{inl}(x_2)) \mid \texttt{inr}(y) \to (y_1, \texttt{inr}(y_2)))$

(8) $((\exists x)(Px \vee Qx)) \Rightarrow ((\exists x)Px \vee (\exists x)Qx)$:
The evidence is $\lambda z.(\texttt{case } z_2 \texttt{ of inl}(x) \to \texttt{inl}(z_1, x) \mid \texttt{inr}(y) \to \texttt{inr}(z_1, y))$

(9) $(\exists x)(Px \Rightarrow (\forall y)Py)$:
The evidence must be a pair (a,f) where $a{:}\mathbb{U}$ and f is a function that takes evidence p for Pa as input and generates a function $g : (x{:}\mathbb{U} \to [Px])$. There is

no way to construct the generic "evidence function" g solely on the basis of the evidence for a specific Pa.

(10) $(\forall x)((\forall y)Py \Rightarrow Px)$: The evidence is $\lambda x.(\lambda f.fx)$

(11) $(\exists x)((\exists y)Py \Rightarrow Px)$:
The evidence must be a pair (a,f) where $a{:}\mathbb{U}$ and f is a function that takes a pair $(b,p) : (y{:}\mathbb{U}\times([Py])$ as input and generates evidence for Pa. The only way to construct f is to make sure that a matches its input b – then the evidence for Pa would be the second input component. There is no way to do this without knowing the input.

(12) $\neg((\exists x)Px) \Rightarrow ((\forall x)(((\exists y)Py) \Rightarrow Px))$: The evidence is $\lambda f.(\lambda x.(\lambda z.\mathtt{any}(fz)))$

(13) $((\exists x)Px) \Rightarrow ((\forall x)(Px \Rightarrow Qx) \Rightarrow ((\exists y)Qy))$: The evidence is $\lambda z.(\lambda f.(z_1,(f\,z_1)\,z_2))$

(14) $\neg((\exists x)Px) \Rightarrow ((\forall x)\neg(Px))$:
The evidence must be a function that takes as input a function $f : (x{:}\mathbb{U}\times[Px])\to\{\}$ and produces a function $g : (x{:}\mathbb{U}\to([Px]\to\{\}))$. To construct g we take an input $x{:}\mathbb{U}$ and evidence $p : [Px]$ and create the element of $\{\}$ by applying f to the pair (x,p). Thus the overall evidence is $\lambda f.(\lambda x.(\lambda p.f(x,p)))$

(15) $((\forall x)\neg(Px)) \Rightarrow \neg((\exists x)Px)$: The evidence is $\lambda f.(\lambda z.(fz_1)z_2)$

(16) $((\exists x)Px) \Rightarrow \neg((\forall x)\neg(Px))$: The evidence is $\lambda z.(\lambda f.(fz_1)z_2)$

(17) $\neg((\forall x)Px) \Rightarrow ((\exists x)\neg(Px))$:
The evidence must be a function that takes as input a function $f : (x{:}\mathbb{U}\to[Px])\to\{\}$ and produces a pair $z : (x{:}\mathbb{U}\times([Px]\to\{\}))$. To construct g we have to find an element $a{:}\mathbb{U}$ and evidence p for Pa. There is no uniform way to construct these two evidences solely from f.

8.4 First-Order refinement proofs and evidence construction

Similar to evidence semantics, the proof calculus of refinement logic can be extended from propositional to first-order logic. Most rules follow immediately from an intuitive understanding of the logical operators. As in our discussion of evidence we investigate each connective separately.

- Refinement rules for atomic formulas
  An atomic formula $A = Pc_1..c_n$ cannot be decomposed into smaller components because it is a placeholder for an unknown proposition. But if A is also one of the assumptions in the hypotheses, we can prove A it using the `axiom` rule.
  $H, a{:}A, H' \vdash A$ `ev` $= a$ `by axiom`
- Refinement rules for $A \Rightarrow B$, $A \wedge B$, $A \vee B$, $\neg A$ are the refinement rules for implications, conjunctions, disjunctions, and negations that we already know from propositional logic. They are summarized in the upper part of table 8.2.
- Refinement rules for $(\forall x)B$
  To prove a universally quantified formula $(\forall x)B$ we have to be able to prove B for every possible instance of the variable x. The only way to do this formally is

to come up with a generic proof for B that does not depend on how x is being instantiated. For this purpose we substitute x by a *new* parameter a' that does not occur elsewhere in the sequent and prove $B[a'/x]$. The fact that a' is new makes sure that nothing is known about a' in the proof of $B[a'/x]$, which means that the proof will in fact apply to every possible instance of x.[5]

$H \vdash (\forall x)B$ `by allR`
$H \vdash B[a'/x]$

To describe the evidence constructed by this rule we assume that we have some evidence b for the the subgoal sequent $H \vdash B[a'/x]$. This means that we must have found a generic way to generate the evidence b for $B[a'/x]$ for arbitrary a'. In other words, there is a function that on input a' computes b and this function must be the evidence for $(\forall x)B$. Using the λ-notation for evidence terms introduced previously, we can write the rule as

$H \vdash (\forall x)B$ $\quad \mathtt{ev} = \lambda a'.b$ `by allR`
$H \vdash B[a'/x]$ $\quad \mathtt{ev} = b$

In addition to refining a universal quantifier in the conclusion of a sequent we also need rules for decomposing quantifiers in assumptions. To prove a goal C by decomposing the assumption $(\forall x)B$ we introduce the assumption $B[a/x]$ for some parameter a and prove C on that basis.

For the construction of evidence let us assume that f is a label for $(\forall x)B$ in the main goal, that b is a label for $B[a/x]$ in the subgoal, and that c the evidence for C. The latter means that there is a function that computes evidence c for C from arbitrary evidences b for $B[a/x]$. Since $f(a)$ is a specific evidence for $B[a/x]$ we can apply this function, i.e. $\lambda b.c$, to $f(a)$, evaluate it and get the desired evidence for C in the main goal. If we integrate evidence construction into the rule we get

$H, f{:}(\forall x)B, H' \vdash C$ $\quad \mathtt{ev} = c[f(a)/b]$ `by allL` a
$H, f{:}(\forall x)B, b{:}B[a/x], H' \vdash C$ $\quad \mathtt{ev} = c$

Note that the parameter a has to be provided when executing the rule since the parameter that instantiates the formula $(\forall x)B$ is not determined by the context of the rule. The user applying the rule has to choose it.

Let us look at two simple concrete examples.

– $(\forall x)(Px \Rightarrow Px)$: The refinement proof is straightforward.

```
⊢ (∀x)(Px⇒Px)      ev = λa.(λp.p)  by allR
1 ⊢ Pa⇒Pa          ev = λp.p       by impliesR
1.1 p:Pa ⊢ Pa      ev = p          by axiom
```

– $((\forall x)Px) \Rightarrow Pa$: The obvious choice for instantiating x is the parameter a.

```
⊢ ((∀x)Px)⇒Pa                ev = λf.f(a)  by impliesR
1 f:(∀x)Px ⊢ Pa              ev = p[f(a)/p] by allL a
1.1 f:(∀x)Px, p:Pa ⊢ Pa      ev = p         by axiom
```

In both cases the constructed evidence is the same as we had in section 8.3.

Note that the rule `allL` explicitly re-introduces the assumption $(\forall x)B$ in the subgoal. The reason for this is that univerally quantified formulas in the assumptions may have to be instantiated several times in order to complete the proof. Any

[5] In our description of the formal rules we use the stroke only to emphasize the fact that a' is new, which is a side condition on the parameters that may actually be chosen when the rule is applied. Apart from that condition, any parameter may be chosen.

proof of $((\forall x)\,Px) \Rightarrow (Pa \wedge Pb)$ must instantiate the variable x with both a and b. If `allL` would drop the assumption $(\forall x)B$, then some proof attempts would not succeed as they cannot show both Pa and Pb. Here is a proof of that statement in refinement logic.

$\vdash$	$((\forall x)Px) \Rightarrow (Pa \wedge Pb)$	$\mathtt{ev} = \lambda f.(f(a), f(b))$	`by impliesR`
`1`	$f{:}(\forall x)Px \vdash Pa \wedge Pb$	$\mathtt{ev} = (f(a), f(b))$	`by allL` a
`1.1`	$f{:}(\forall x)Px, p_a{:}Pa \vdash Pa \wedge Pb$	$\mathtt{ev} = (p_a, f(b))$	`by allL` b
`1.1.1`	$f{:}(\forall x)Px, p_a{:}Pa, p_b{:}Pb \vdash Pa \wedge Pb$	$\mathtt{ev} = (p_a, p_b)$	`by andR`
`1.1.1.1`	$f{:}(\forall x)Px, p_a{:}Pa, p_b{:}Pb \vdash Pa$	$\mathtt{ev} = p_a$	`by axiom`
`1.1.1.2`	$f{:}(\forall x)Px, p_a{:}Pa, p_b{:}Pb \vdash Pb$	$\mathtt{ev} = p_b$	`by axiom`

- Refinement rules for $(\exists x)B$

  To prove an existentially quantified formula $(\exists x)B$ we have to be able to prove $B[a/x]$ for some parameter a. If b is the evidence for $B[a/x]$ then combining this evidence with the parameter a is evidence for the existence of an x such that B holds.

$H \vdash (\exists x)B$	$\mathtt{ev} = (a, b)$	`by exR` a
$H \vdash B[a/x]$	$\mathtt{ev} = b$	

  Again, the parameter a has to be provided when executing the rule.

  To prove a goal C by decomposing the assumption $(\exists x)B$ we introduce the assumption $B[a'/x]$ for a new parameter a' and prove C on that basis. As in the case of `allR` requiring a' to be new makes sure that nothing is known about a' when we use $B[a'/x]$ to prove C except that a' does exist. For the construction of evidence let us assume that z is a label for $(\exists x)B$ in the main goal, b is the evidence for the assumption $B[a/x]$ in the subgoal, and c the evidence for C. Then z is assumed to be the same as the evidence pair (a', b), which means that replacing a by z_1 and b by z_2 in c gives us the evidence for the main goal.

$H, z{:}(\exists x)B, H' \vdash C$	$\mathtt{ev} = c[z_1, z_2/a', b]$	`by exL`
$H, b{:}B[a'/x], H' \vdash C$	$\mathtt{ev} = c$	

  Again let us look at a few concrete examples

  – $Pa \Rightarrow ((\exists x)Px)$: Again a simple and straightforward proof

$\vdash$	$Pa \Rightarrow ((\exists x)Px)$	$\mathtt{ev} = \lambda p.(a, p)$	`by impliesR`
`1`	$p{:}Pa \vdash (\exists x)Px$	$\mathtt{ev} = (a, p)$	`by exR` a
`1.1`	$p{:}Pa \vdash Pa$	$\mathtt{ev} = p$	`by axiom`

  – $((\exists x)Px) \Rightarrow ((\exists y)Py)$:

$\vdash$	$((\exists x)Px) \Rightarrow ((\exists y)Py)$	$\mathtt{ev} = \lambda z.(z_1, z_2)$	`by impliesR`
`1`	$z{:}(\exists x)Px \vdash (\exists y)Py$	$\mathtt{ev} = (z_1, z_2)$	`by exL`
`1.1`	$p{:}Pa \vdash (\exists y)Py$	$\mathtt{ev} = (a, p)$	`by exR` a
`1.1.1`	$p{:}Pa \vdash Pa$	$\mathtt{ev} = p$	`by axiom`

  Note that (z_1, z_2) is idential to z so the evidence constructed is the same as in section 8.3. Note also that the order of rule applications is important. Had we used the rule `exR` a first we wouldn't be able to complete the proof.

$\vdash$	$((\exists x)Px) \Rightarrow ((\exists y)Py)$	`by impliesR`
`1`	$z{:}(\exists x)Px \vdash (\exists y)Py$	`by exR` a
`1.1`	$z{:}(\exists x)Px \vdash Pa$	`by ???`

Table 8.2 summarizes all the rules of the refinement calculus for first order logic.

left		right	
impliesL	$H, f{:}A \Rightarrow B, H' \vdash C$ $\quad \mathtt{ev} = c[f(a)/b]$ $H, f{:}A \Rightarrow B, H' \vdash A \quad \mathtt{ev} = a$ $H, b{:}B, H' \vdash C \quad \mathtt{ev} = c$	$H \vdash A \Rightarrow B \quad \mathtt{ev} = \lambda a.b$ $H, a{:}A \vdash B \quad \mathtt{ev} = b$	impliesR
andL	$H, x{:}A \wedge B, H' \vdash C \quad \mathtt{ev} = c[x_1, x_2/a, b]$ $H, a{:}A, b{:}B, H' \vdash C \quad \mathtt{ev} = c$	$H \vdash A \wedge B \quad \mathtt{ev} = (a, b)$ $H \vdash A \quad \mathtt{ev} = a$ $H \vdash B \quad \mathtt{ev} = b$	andR
orL	$H, x{:}A \vee B, H' \vdash C \quad \mathtt{ev} = \mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}(a) \to c_1 \mid \mathtt{inr}(b) \to c_2$ $H, a{:}A, H' \vdash C \quad \mathtt{ev} = c_1$ $H, b{:}B, H' \vdash C \quad \mathtt{ev} = c_2$	$H \vdash A \vee B \quad \mathtt{ev} = \mathtt{inl}(a)$ $H \vdash A \quad \mathtt{ev} = a$ $H \vdash A \vee B \quad \mathtt{ev} = \mathtt{inr}(b)$ $H \vdash B \quad \mathtt{ev} = b$	orR1 orR2
notL	$H, f{:}\neg A, H' \vdash C \quad \mathtt{ev} = \mathtt{any}(f(a))$ $H, f{:}\neg A, H' \vdash A \quad \mathtt{ev} = a$	$H \vdash \neg A \quad \mathtt{ev} = \lambda a.b$ $H, a{:}A \vdash \mathtt{f} \quad \mathtt{ev} = b$	notR
		$H, a{:}A, H' \vdash A \quad \mathtt{ev} = a$	axiom
allL a	$H, f{:}(\forall x)B, H' \vdash C \quad \mathtt{ev} = c[f(a)/b]$ $H, f{:}(\forall x)B, b{:}B[a/x], H' \vdash C \quad \mathtt{ev} = c$	$H \vdash (\forall x)B \quad \mathtt{ev} = \lambda a'.b$ $H \vdash B[a'/x] \quad \mathtt{ev} = b$	allR
exL	$H, z{:}(\exists x)B, H' \vdash C \quad \mathtt{ev} = c[z_1, z_2/a', b]$ $H, b{:}B[a'/x], H' \vdash C \quad \mathtt{ev} = c$	$H \vdash (\exists x)B \quad \mathtt{ev} = (a, b)$ $H \vdash B[a/x] \quad \mathtt{ev} = b$	exR a
a can be an arbitrary parameter while a' must be new			

Table 8.2 Rules of the first-order refinement calculus

Exercises

As an exercise the following problems should be investigated in groups. For each of the formulas below the group should find a refinement proof and construct the evidence from the proof. In cases where no proof can be found, try to explain why the proof has to get stuck.

(1) $((\forall x)(Px \wedge Qx)) \Rightarrow ((\forall x)Px \wedge (\forall x)Qx)$:

$\vdash ((\forall x)(Px \wedge Qx)) \Rightarrow ((\forall x)Px \wedge (\forall x)Qx)$ by impliesR
1 $(\forall x)(Px \wedge Qx) \vdash ((\forall x)Px \wedge (\forall x)Qx)$ by andR
1.1 $(\forall x)(Px \wedge Qx) \vdash (\forall x)Px$ by allR
1.1.1 $(\forall x)(Px \wedge Qx) \vdash Pa$ by allL a
1.1.1.1 $(\forall x)(Px \wedge Qx), Pa \wedge Qa \vdash Pa$ by andL
1.1.1.1.1 $(\forall x)(Px \wedge Qx), Pa, Qa \vdash Pa$ by axiom
1.2 $(\forall x)(Px \wedge Qx) \vdash (\forall x)Qx$ by allR
1.2.1 $(\forall x)(Px \wedge Qx) \vdash Qa$ by allL a
1.2.1.1 $(\forall x)(Px \wedge Qx), Pa \wedge Qa \vdash Qa$ by andL
1.2.1.1.1 $(\forall x)(Px \wedge Qx), Pa, Qa \vdash Qa$ by axiom

The evidence extracted from this proof is $\lambda f.(\lambda x.(f\,x)_1, \lambda x.(f\,x)_2)$

(2) $((\forall x)Px \wedge (\forall x)Qx) \Rightarrow ((\forall x)(Px \wedge Qx))$:

$\vdash ((\forall x)Px \wedge (\forall x)Qx) \Rightarrow ((\forall x)(Px \wedge Qx))$ by impliesR
1 $(\forall x)Px \wedge (\forall x)Qx \vdash (\forall x)(Px \wedge Qx)$ by andL
1.1 $(\forall x)Px, (\forall x)Qx \vdash (\forall x)(Px \wedge Qx)$ by allR
1.1.1 $(\forall x)Px, (\forall x)Qx \vdash Pa \wedge Qa$ by andR
1.1.1.1 $(\forall x)Px, (\forall x)Qx \vdash Pa$ by allL a
1.1.1.1.1 $(\forall x)Px, (\forall x)Qx, Pa \vdash Pa$ by axiom
1.1.1.2 $(\forall x)Px, (\forall x)Qx \vdash Qa$ by allL a
1.1.1.2.1 $(\forall x)Px, (\forall x)Qx, Qa \vdash Qa$ by axiom

(3) $((\forall x)Px \vee (\forall x)Qx) \Rightarrow ((\forall x)(Px \vee Qx))$:

$\vdash ((\forall x)Px \vee (\forall x)Qx) \Rightarrow ((\forall x)(Px \vee Qx))$	by impliesR
1 $(\forall x)Px \vee (\forall x)Qx \vdash (\forall x)(Px \vee Qx)$	by orL
1.1 $(\forall x)Px \vee (\forall x)Qx \vdash Pa \vee Qa$	by allR
1.1.1 $(\forall x)Px \vdash Pa \vee Qa$	by allL a
1.1.1.1 $Pa \vdash Pa \vee Qa$	by orR1
1.1.1.1.1 $Pa \vdash Pa$	by axiom
1.1.2 $(\forall x)Qx \vdash Pa \vee Qa$	by
1.1.2.1 $Qa \vdash Pa \vee Qa$	by orR2
1.1.2.1.1 $Qa \vdash Qa$	by axiom

(4) $((\forall x)(Px \vee Qx)) \Rightarrow ((\forall x)Px \vee (\forall x)Qx)$: A proof attempt will get stuck

$\vdash ((\forall x)(Px \vee Qx)) \Rightarrow ((\forall x)Px \vee (\forall x)Qx)$	by impliesR
1 $(\forall x)(Px \vee Qx) \vdash (\forall x)Px \vee (\forall x)Qx$	by ???

At this point we have to prove either $(\forall x)Px$ or $(\forall x)Qx$ but there is no way to prove that.

(5) $((\exists x)(Px \wedge Qx)) \Rightarrow ((\exists x)Px \wedge (\exists x)Qx)$:

$\vdash ((\exists x)(Px \wedge Qx)) \Rightarrow ((\exists x)Px \wedge (\exists x)Qx)$	by impliesR
1 $(\exists x)(Px \wedge Qx) \vdash (\exists x)Px \wedge (\exists x)Qx$	by exL
1.1 $Pa \wedge Qa \vdash (\exists x)Px \wedge (\exists x)Qx$	by andL
1.1.1 $Pa, Qa \vdash (\exists x)Px \wedge (\exists x)Qx$	by andR
1.1.1.1 $Pa, Qa \vdash (\exists x)Px$	by exR a
1.1.1.1.1 $Pa, Qa \vdash Pa$	by axiom
1.1.1.2 $Pa, Qa \vdash (\exists x)Qx$	by exR a
1.1.1.2.1 $Pa, Qa \vdash Qa$	by axiom

(6) $((\exists x)Px \wedge (\exists x)Qx) \Rightarrow ((\exists x)(Px \wedge Qx))$: Here is a proof attempt

```
⊢ ((∃x)Px∧(∃x)Qx)⇒((∃x)(Px∧Qx))   by impliesR
1   (∃x)Px∧(∃x)Qx ⊢ (∃x)(Px∧Qx)   by andL
1.1   (∃x)Px,(∃x)Qx ⊢ (∃x)(Px∧Qx)   by exL
1.1.1   Pa,(∃x)Qx ⊢ (∃x)(Px∧Qx)   by exL
1.1.1   Pa,Qb ⊢ (∃x)(Px∧Qx)   by ???
```

The proof gets stuck because in the second application of `exL` we will have to use a *new* parameter instead of using a again.

(7) $((\exists x)Px \vee (\exists x)Qx) \Rightarrow ((\exists x)(Px \vee Qx))$:

```
⊢ ((∃x)Px∨(∃x)Qx)⇒((∃x)(Px∨Qx))   by impliesR
1   (∃x)Px∨(∃x)Qx ⊢ (∃x)(Px∨Qx)   by orL
1.1   (∃x)Px ⊢ (∃x)(Px∨Qx)   by exL
1.1.1   Pa ⊢ (∃x)(Px∨Qx)   by exR a
1.1.1.1   Pa ⊢ Pa∨Qa   by orR1
1.1.1.1.1   Pa ⊢ Pa   by axiom
1.2   (∃x)Qx ⊢ (∃x)(Px∨Qx)   by exL
1.2.1   Qa ⊢ (∃x)(Px∨Qx)   by exR a
1.2.1.1   Qa ⊢ Pa∨Qa   by orR2
1.2.1.1.1   Qa ⊢ Qa   by axiom
```

(8) $((\exists x)(Px \vee Qx)) \Rightarrow ((\exists x)Px \vee (\exists x)Qx)$:

```
⊢ ((∃x)(Px∨Qx))⇒((∃x)Px∨(∃x)Qx)   by impliesR
1   (∃x)(Px∨Qx) ⊢ (∃x)Px∨(∃x)Qx   by exL
1.1   Pa∨Qa ⊢ (∃x)Px∨(∃x)Qx   by orL
1.1.1   Pa ⊢ (∃x)Px∨(∃x)Qx   by orR1
1.1.1.1   Pa ⊢ (∃x)Px   by exR a
1.1.1.1.1   Pa ⊢ Pa   by axiom
1.1.2   Pa ⊢ (∃x)Px∨(∃x)Qx   by orR1
1.1.2.1   Pa ⊢ (∃x)Px   by exR a
1.1.2.1.1   Pa ⊢ Pa   by axiom
```

(9) $(\exists x)(Px \Rightarrow (\forall y)Py)$: Here is a proof attempt

```
⊢ (∃x)(Px⇒(∀y)Py)   by exR a
1   ⊢ Pa⇒(∀y)Py   by impliesR
1.1   Pa ⊢ (∀y)Py   by allR
1.1.1   Pa ⊢ Pb   by ???
```

The proof gets stuck because in the application of `allR` we will have to use a *new* parameter instead of using a again.

(10) $(\forall x)((\forall y)Py \Rightarrow Px)$:

```
⊢ (∀x)((∀y)Py⇒Px)   by allR
1   ⊢ (∀y)Py⇒Pa   by impliesR
1.1   (∀y)Py ⊢ Pa   by allL a
1.1.1   Pa ⊢ Pa   by axiom
```

(11) $(\exists x)((\exists y)Py \Rightarrow Px)$: Here is a proof attempt

```
⊢ (∃x)((∃y)Py⇒Px)   by exR a
1   ⊢ (∃y)Py⇒Pa   by impliesR
1.1   (∃y)Py ⊢ Pa   by exL
1.1.1   Pb ⊢ Pa   by ???
```

The proof gets stuck because in the application of `exL` we will have to use a *new* parameter instead of using a again.

(12) $\neg((\exists x)Px) \Rightarrow ((\forall x)(((\exists y)Py) \Rightarrow Px))$:

```
⊢ ¬((∃x)Px)⇒((∀x)(((∃y)Py)⇒Px))   by impliesR
1   ¬((∃x)Px) ⊢ (∀x)(((∃y)Py)⇒Px)   by allR
```

```
1.1   ¬((∃x)Px) ⊢ ((∃y)Py)⇒Pa      by impliesR
1.1.1   ¬((∃x)Px),(∃y)Py ⊢ Pa      by notL
1.1.1.1   ¬((∃x)Px),(∃y)Py ⊢ (∃x)Px by axiom
```

(13) $((\exists x)Px)\Rightarrow((\forall x)(Px\Rightarrow Qx)\Rightarrow((\exists y)Qy))$:

```
⊢ ((∃x)Px)⇒((∀x)(Px⇒Qx)⇒((∃y)Qy)) by impliesR
1   (∃x)Px ⊢ (∀x)(Px⇒Qx)⇒((∃y)Qy) by impliesR
1.1   (∃x)Px,(∀x)(Px⇒Qx) ⊢ (∃y)Qy by exL
1.1.1   Pa,(∀x)(Px⇒Qx) ⊢ (∃y)Qy   by allL a
1.1.1.1   Pa,Pa⇒Qa ⊢ (∃y)Qy       by exR a
1.1.1.1.1   Pa,Pa⇒Qa ⊢ Qa         by impliesL
1.1.1.1.1.1   Pa,Pa⇒Qa ⊢ Pa       by axiom
1.1.1.1.1.2   Pa,Pa⇒Qa,Qa ⊢ Qa   by axiom
```

(14) $\neg((\exists x)Px)\Rightarrow((\forall x)\neg(Px))$:

```
⊢ ¬((∃x)Px)⇒((∀x)¬(Px))           by impliesR
1   ¬((∃x)Px) ⊢ (∀x)¬(Px)         by allR
1.1   ¬((∃x)Px) ⊢ ¬(Pa)           by notR
1.1.1   ¬((∃x)Px),Pa ⊢ f          by notL
1.1.1.1   ¬((∃x)Px),Pa ⊢ (∃x)Px   by exR a
1.1.1.1.1   ¬((∃x)Px),Pa ⊢ Pa     by axiom
```

(15) $((\forall x)\neg(Px))\Rightarrow\neg((\exists x)Px)$:

```
⊢ ((∀x)¬(Px))⇒¬((∃x)Px)           by impliesR
1   (∀x)¬(Px)) ⊢ ¬((∃x)Px)        by notR
1.1   (∀x)¬(Px)),(∃x)Px ⊢ f       by exL
1.1.1   (∀x)¬(Px)),Pa ⊢ f         by allL a
1.1.1.1   ¬(Pa),Pa ⊢ f            by notL
1.1.1.1.1   ¬(Pa),Pa ⊢ Pa         by axiom
```

(16) $((\exists x)Px)\Rightarrow\neg((\forall x)\neg(Px))$:

```
⊢ ((∃x)Px)⇒¬((∀x)¬(Px)            by impliesR
1   (∃x)Px ⊢ ¬((∀x)¬(Px)          by exL
1.1   Pa ⊢ ¬((∀x)¬(Px)            by notR
1.1.1   Pa,(∀x)¬(Px) ⊢ f          by allL a
1.1.1.1   Pa,¬(Pa) ⊢ f            by notL
1.1.1.1.1   Pa,¬(Pa) ⊢ Pa         by axiom
```

(17) $\neg((\forall x)Px)\Rightarrow((\exists x)\neg(Px))$: Here is a proof attempt

```
⊢ ¬((∀x)Px)⇒((∃x)¬(Px)))          by impliesR
1   ⊢   ¬((∀x)Px) ⊢ ((∃x)¬(Px)))  by ???
```

At this point we're stuck. If we apply `notL` we will lose the conclusion $((\exists x)\neg(Px)))$ and have to prove $(\forall x)Px$, which clearly won't work. But there are no other proof rule that can be applied here.

Chapter 9
First-Order Proof Systems

9.1 Boolean Semantics

The boolean semantics of first-order logic, like the one of propositional logic, is based on a concept of valuations. In propositional logic, it was sufficient to assign values to all propositional variables and then extend the evaluation from atoms to formulas in a canonical fashion by showing how to calculate the value of a composed formula from values of the subformulas. In first-order logic, our starting point has to be atomic formulas instead of propositional variables and we have to explain how to calculate the value of quantified formulas that may have infinitely many subformulas.

The standard approach is to interpret parameters by elements of some *universe* $\mathscr{U}$ and n-ary predicates by subsets of $\mathscr{U}^n$. A closed formula $Pa_1..a_n$ then expresses the fact that the interpretations $k_i \in \mathscr{U}$ of the a_i, taken together as n-tuple $(k_1,..,k_n)$, form an element of the interpretation of P.

Smullyan's approach is similar but avoids set theory altogether. Instead, he introduces *$\mathscr{U}$-formulas*, where the elements of the universe $\mathscr{U}$ are used as parameters and defines *first-order valuations* as canonical extensions of boolean valuations on the set $E^{\mathscr{U}}$ of all closed $\mathscr{U}$-formulas. The semantics of arbitrary formulas is then defined by a mapping φ from the set of parameters into $\mathscr{U}$.

Definition 9.1 (first-order valuations).
A first-order valuation v of $E^{\mathscr{U}}$ is an assignment of truth values to elements of $E^{\mathscr{U}}$ such that

(1) v is a boolean valuation of $E^{\mathscr{U}}$, i.e.
$v[\neg A] = \mathrm{t}$ iff $v[A] = \mathrm{f}$
$v[A \wedge B] = \mathrm{t}$ iff $v[A] = \mathrm{t}$ and $v[B] = \mathrm{t}$
$v[A \vee B] = \mathrm{t}$ iff $v[A] = \mathrm{t}$ or $v[B] = \mathrm{t}$
$v[A \Rightarrow B] = \mathrm{t}$ iff $v[A] = \mathrm{f}$ or $v[B] = \mathrm{t}$

(2) $v[(\forall x)B] = \mathrm{t}$ iff $v[B|^x_k] = \mathrm{t}$ for every $k \in \mathscr{U}$
$v[(\exists x)B] = \mathrm{t}$ iff $v[B|^x_k] = \mathrm{t}$ for at least one $k \in \mathscr{U}$

All valuations can be defined as canonical extensions of *atomic valuations*, i.e. assignments of truth values to the atomic formulas in $E^{\mathscr{U}}$. A *valuation tree* for a formula A is the formation tree of A together with a consistent assignment of truth values to all the nodes in that tree.

Note that since formation trees are usually infinite, one cannot expect to *compute* the truth value of a formula A solely on the basis of a given atomic valuation.

As in propositional logic, the semantics of formulas can also be described via via truth sets.

Definition 9.2. A first-order truth set S (w.r.t. $\mathscr{U}$) is a subset of of $E^{\mathscr{U}}$ such that

(1) S statisfies the requirements on propositional truth sets, i.e.

$A \in S$ iff $\neg A \notin S$

$A \wedge B \in S$ iff $A \in S$ and $B \in S$

$A \vee B \in S$ iff $A \in S$ or $B \in S$

$A \Rightarrow B \in S$ iff $A \notin S$ or $B \in S$

(2) $(\forall x)B \in S$ iff $B|^x_k \in S$ for every $k \in \mathscr{U}$

$(\exists x)B \in S$ iff $B|^x_k \in S$ for at least one $k \in \mathscr{U}$

It is easy (though tedious) to show that truth sets correspond to valuations in the sense that every first-order truth set is exactly the set of all formulas that are true under a fixed first-order valuation.

The definition of first-order valuations can be extended to sentences with parameters as follows. Let φ be a mapping from the set of parameters to $\mathscr{U}$. For a formula A define A^{φ} to be the result of replacing every parameter a_i in A by $\varphi(a_i)$. We say that A is *true under φ and v* if $v[A^{\varphi}] = t$.

The standard semantics of first-order formulas can be linked to the above as follows. Let E define the set of all closed formulas. An *interpretation of E* is a triple $I = (\mathscr{U}, \varphi, \iota)$, where $\mathscr{U}$ is an arbitrary set, φ is a mapping from the set of parameters to $\mathscr{U}$, and ι is a function that maps each n-ary predicate symbol P to a set $I(P) \subseteq \mathscr{U}^n$ (or an n-ary relation over $\mathscr{U}$).

An atomic sentence $Pa_1..a_n$ is *true under I* if $(\varphi(a_1),..\varphi(a_n)) \in \iota(P)$. In this manner, every interpretation induces an atomic valuation v_0 (together with φ), defined by $v_0[Pa_1..a_n^{\varphi}] = t$ iff $(\varphi(a_1),..\varphi(a_n)) \in \iota(P)$, and vice versa ($\iota(P) = \{\varphi(a_1),..\varphi(a_n)) \mid v_0[Pa_1..a_n^{\varphi}] = t\}$). From now on we will use whatever notion is more convenient.

A formula A is called *satisfiable* if it is true under at least one interpretation I (i.e. under at least one universe $\mathscr{U}$, one mapping φ, and one interpretation of the predicate symbols). I is also called a *model* of A. A is *valid* if A is true under every interpretation. These notions can be extended to sets of formula sin a canonical fashion.

It should be noted that there is a fine distinction between boolean valuations and first-order valuations. Boolean valuations can only analyze the propositional structure of formulas. They cannot evaluate quantified formulas and therefore have to treat them like propositional variables. In contrast to that first-order valuations can analyze the internals of quantified formulas and extract information that is unaccessible to boolean valuations.

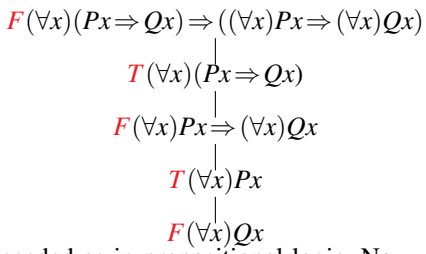
For instance, a boolean valuations would interpret the logical structure of the formula
$(\forall x)(Px \wedge Qx) \Rightarrow (\forall x)Px$ as $PQ \Rightarrow P$, which is obviously not a tautology. In contrast to that, every first-order valuation would go into the details of $(\forall x)(Px \wedge Qx)$ and $(\forall x)Px$ and evaluate to true. Thus the formula is valid, but not a tautology.

For the same reason, the formula $(\forall x)(Px \wedge Qx) \wedge (\exists x)(\neg Px)$ is truth-functionally satisfiable but not first-order satisfiable, since there is no first-order valuation (with a non-empty universe) that can make it true.

First-order valuations provide a more specific analysis than boolean valuations can give. They agree on quantifier-free formulas, however (Exercise!), and in that sense first-order logic is a canonical extension of propositional logic.

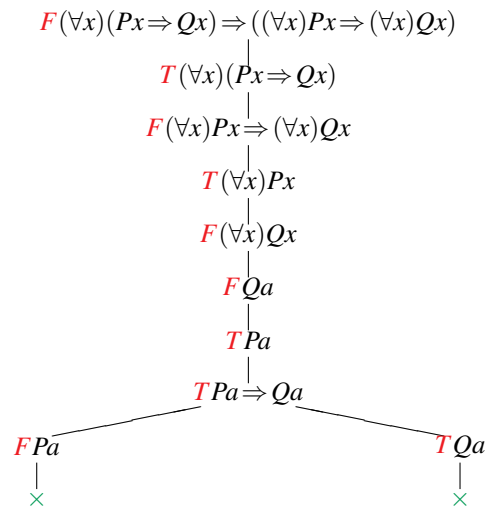
9.2 First-Order Tableaux

Since the evaluation of quantified formulas usually requires the evaluation of the formula for all possible elements of the universe, truth tables are unsuited for proving first-order formulas correct. Universes are usually infinite and even in a finite universe, the search space would quickly explode. The extension of the tableaux method to first-order logic, on the other hand, is quite straightforward. Let us consider an example.



Up to this point we have proceeded as in propositional logic. Now we have to start decomposing quantifiers. The formula $(\forall x)Qx$ is false if Qx can be made false for at least one element k of the universe. Since the elements of the universe do not belong to the syntax of the formulas, we substitute x by a parameter a instead.

In the following step we decompose $T(\forall x)Px$. We know that $(\forall x)Px$ is true if Px is true for all elements of the universe. This means we can substitute any parameter for x and we choose a again, since this is useful for completing the proof. The remaining proof is straightforward and we get



Why did we decompose $F(\forall x)Qx$ before $T(\forall x)Px$ in the proof?

The parameter a that we substituted for x was supposed to indicate that Qx can be made false by some yet unknown element of the universe. Since we do not know this element, a should be a *new parameter* – this way we make sure that we don't

make any further assumptions about a by accidentally linking it to a parameter that was introduced earlier in the proof.

If we were to decompose $T(\forall x)Px$ before $F(\forall x)Qx$ then we would not be able to use a as parameter for Q, since it has already been used for P and is not unknown anymore. If we decompose $F(\forall x)Qx$ first, then a is still new. Choosing the same a for P is a decision we make afterwards.

In informal mathematics, quantifiers are handled in exactly the same way. When proving
$(\forall x)(Px \wedge Qx) \Rightarrow (\forall x)Qx$ we assume $(\forall x)(Px \wedge Qx)$ and then try to show $(\forall x)Qx$. For this purpose we assume a to be arbitrary, but fixed, and try to prove Qa. Since we know $(\forall x)(Px \wedge Qx)$, we also know that $Pa \wedge Qa$ holds for the arbitrary a that we just chose and conclude that Qa is in fact the case. Note that it was crucial to have the a before instantiating $(\forall x)(Px \wedge Qx)$.

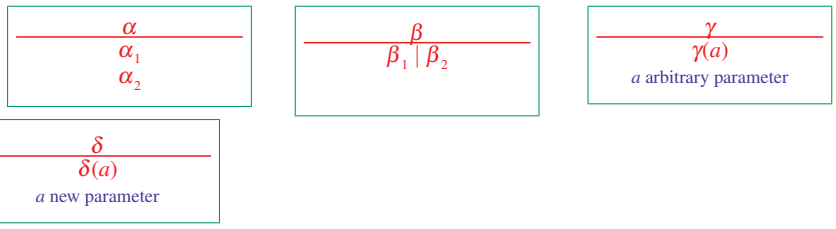
9.3 Extension of the unified notation

The above example shows that there are two different ways to handle quantifiers in tableaux proofs.

In the first case, we have formulas of the form $T(\forall x)A$ and, by duality, $F(\exists x)A$, which we call formulas of type γ of *universal type*. γ-formulas are decomposed into $TB[a/x]$ (and $FB[a/x]$, respectively), where a is an arbitrary parameter. These formulas are often denoted by $\gamma(a)$.

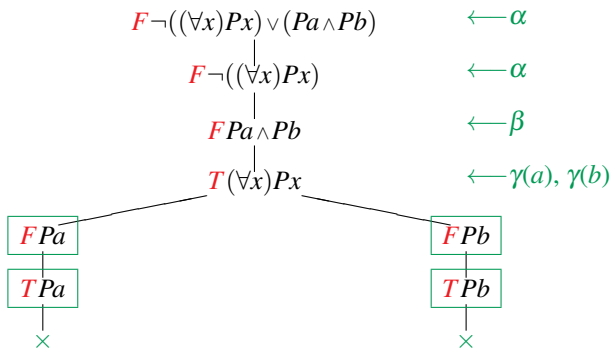
In the other case, we have formulas of the form $F(\forall x)A$ and, by duality, $T(\exists x)A$, which we call formulas of type δ of *existential type*. δ-formulas are decomposed into $FB[a/x]$ (and $TB[a/x]$, respectively), where a is a new parameter. These formulas are often denoted by $\delta(a)$ and the requirement that a must be new is usually called the *proviso* of the rule.

Altogether we have now four types of inference rules.[1]



Here is another example proof

[1] In calculi that use terms instead of parameters, the γ-rule allows a to be an arbitrary term (representing some object) whereas in the δ rule a must be a new variable, representing the fact that the element of the universe is unknow.



Note that in this proof, the γ-formula $T(\forall x)Px$ had to be instantiated twice to complete the proof. In general, formulas of universal type may be used arbitrarily often in a proof and therefore validity in first-order logic is not decidable.[2]

9.4 A liberalized δ rule

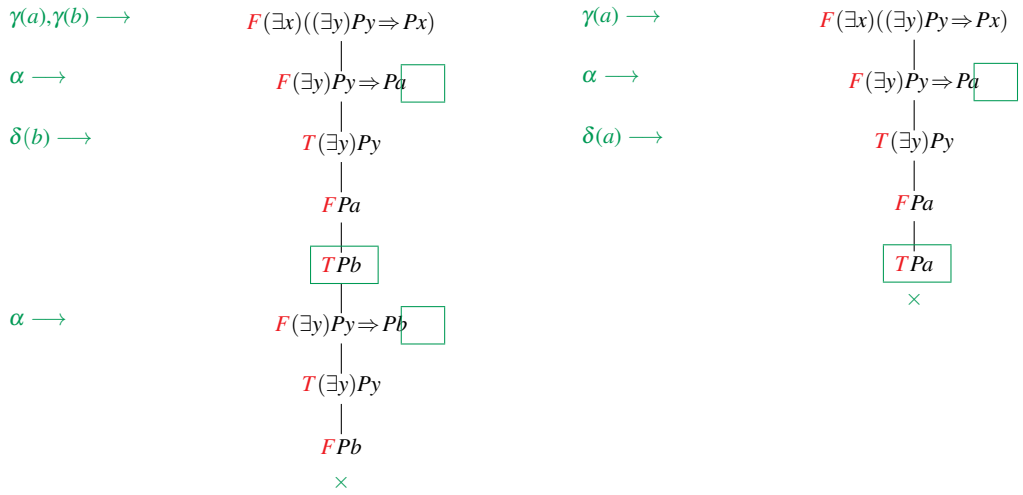
The proviso of the δ rule, which requires a to be a new parameter, is quite restrictive and makes formal proofs more complicated than they have to be. Actually, the proviso is more restrictive than it has to be. It is possible to liberalize the δ rule by replacing it by the following requirement:

> provided a is a new parameter
> or a was not previously introduced on the same path by a δ rule, does not occur in δ, and no parameter in δ was previously generated by a δ rule

In other words, if a does already occur in the proof then we may use it in a δ rule if it was generated by some γ rule. The rationale is that this γ rule could also be applied later and use the parameter a at that point ... after the δ rule has introduced it. Thus the fact that the γ rule appears earlier in the proof should not affect the parameters that the δ rule is permitted to use.

The following example shows the advantages of using a liberalized δ rule. In the proof on the left, the (original) the δ rule, which can only be applied after the first application of the γ rule, cannot use the parameter a because it already occurs in the proof. It has to use a new parameter b instead and we have to apply the γ rule again to get the formula FPb. Using the liberalized δ rule instead makes the proof on the right much shorter.

[2] This argument only appeals to the intuition. The actual proof of the undecidability of first-order logic is more complex, since one has to show that there is no other way to determine that a formula is not valid.



9.5 Transforming tableaux proofs into Refinement logic proofs

In propositional logic we could use booleanization to relate the truth table semantics of tableaux to the evidence semantics of refinement logic. This technique can be extended to first-order logic if the domains under consideration are finite. In this case a universally quantified formula $(\forall x)B$ corresponds to the formula $B[a_1/x]\wedge\ldots\wedge B[a_n/x]$ and a universally quantified formula $(\exists x)B$ to $B[a_1/x]\vee\ldots\vee B[a_n/x]$, where $a_1,..,a_n$ are (representatives of) all the elements of the universe, and the decidability of atomica formulas would propagate to all formulas of first-order logic.

Beyond finite domains all bets are off. We cannot guarantee the decidability of formulas anymore and evidence will be increasingly hard or impossible to construct. To what extent the booleanization of evidence can be extended to first-order formulas in these cases stillneeds to be researched.

If the decidability of formulas can be guaranteed, we can translate tableaux proofs into refinement logic proofs using the same method as in propositional logic. We convert tableaux into block tableaux, convert the block tableaux rules into rules that generate proofs with only one F-formula, and perform a syntax translation that separates the T-formulas from the F-formulas by putting a between them and then drops the signs. This leads to decomposition rules that are mostly identical to the rules of the propositional refinement calculus, except for the rules `orR1,`, `orR2`, `impliesL`, `notL`, and – as the example of $(\exists x)((\exists y)Py\Rightarrow Px)$ in Section 9.4 shows – `exR`.[3] For these rules we provide refinement logic proof fragments that simulate their behavior. The proof fragments for `orR1`*, `orR2`*, `impliesL`*, and `notL`* have already been discussed in our account of propositional logic. The simulation of the rule `exR`* is given below.

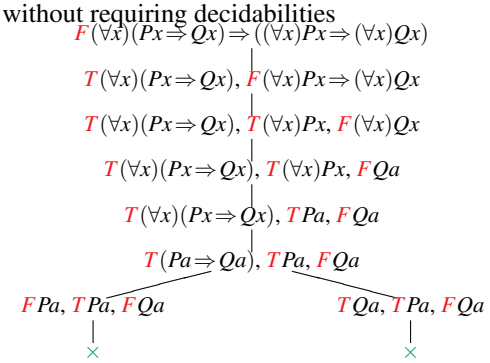
```
exR*:  H ⊢ (∃x)B                          by Use_decidability_of (∃x)B
    1    H, ((∃x)B)∨¬((∃x)B) ⊢ (∃x)B      by orL
    1.1    H, (∃x)B ⊢ (∃x)B                by axiom
```

```
1.2     H, ¬((∃x)B) ⊢ (∃x)B        by exR a
1.2.1     H, ¬((∃x)B) ⊢ B[a/x]
```

All other rules do not introduce additional F-formulas and immediately translat into the corresponding rules of refinement logic. Table 9.1 summarizes the translation of the tableaux rules into refinement rules.

We conclude this section with a few examples.

Example 9.3. (Translation of tableaux roofs into refinement logic)
The block tableau for $(\forall x)(Px \Rightarrow Qx) \Rightarrow ((\forall x)Px \Rightarrow (\forall x)Qx)$ translates immediately without requiring decidabilities




```
⊢ (∀x)(Px⇒Qx)⇒((∀x)Px⇒(∀x)Qx)                    by impliesR
1 (∀x)(Px⇒Qx) ⊢ (∀x)Px⇒(∀x)Qx                    by impliesR
1.1 (∀x)(Px⇒Qx), (∀x)Px ⊢ (∀x)Qx                 by allR
1.1.1 (∀x)(Px⇒Qx), (∀x)Px ⊢ Qa                   by allL a
1.1.1.1 (∀x)(Px⇒Qx), Pa ⊢ Qa                     by allL a
1.1.1.1.1 Pa⇒Qa, Pa ⊢ Qa                         by impliesL
1.1.1.1.1.1 Pa⇒Qa, Pa ⊢ Pa                       by axiom
1.1.1.1.1.2 Pa⇒Qa, Pa ⊢ Pa                       by axiom
```

[3] In principle, all signed formulas may be decomposed multiple times in a tableau proof, since the tableaux calculus does not explicitly exclude that. But this option has a significant effect only in the case of the γ-rules, where it enables us to instantiate the γ-subformulas several times with different parameters. As a result the $T\forall$-rule will create multiple T-formulas, which is captured in the `allL` rule for refinement logic, and the $F\exists$-rule will create multiple F-formulas, which is not permitted for `exR`.

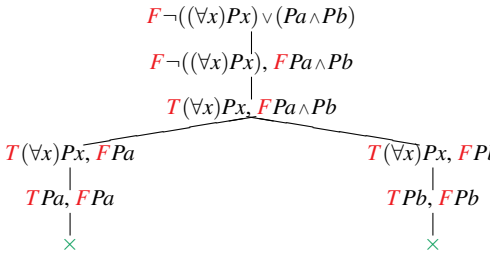
	T	F	
α	$S, TA \wedge B$ S, TA, TB	$S, FA \wedge B$ S, FA S, FB	β
β	$S, TA \vee B$ S, TA S, TB	$S, FA \vee B$ $S, FA\ ,\ FB$	α
β	$S, TA \Rightarrow B$ S, FA S, TB	$S, FA \Rightarrow B$ S, TA, FB	α
α	$S, T\neg A$ S, FA	$S, F\neg A$ S, TA	α
$*$	S, TA, FA		
δ	$S, T(\exists x)B$ $S, TB[a'/x]$	$S, F(\exists x)B$ $S, FB[a/x]$	γ
γ	$S, T(\forall x)B$ $S, TB[a/x]$	$S, F(\forall x)B$ $S, FB[a'/x]$	δ

$\mapsto$

	left	right	
andL	$S, A \wedge B \vdash C$ $S, A, B \vdash C$	$S \vdash A \wedge B$ $S \vdash A$ $S \vdash B$	andR
orL	$S, A \vee B \vdash C$ $S, A \vdash C$ $S, B \vdash C$	$S \vdash A \vee B$ $S, \neg A \vdash \mathrm{B}$ $S \vdash A \vee B$ $S, \neg B \vdash A$	orR1* orR2*
impliesL*	$S, A \Rightarrow B \vdash C$ $S, \neg C \vdash A$ $S, B \vdash C$	$S \vdash A \Rightarrow B$ $S, A \vdash B$	impliesR
notL*	$S, \neg A \vdash C$ $S, \neg C \vdash A$	$S \vdash \neg A$ $S, A \vdash \mathrm{f}$	notR
axiom	$S, A \vdash A$		
exL	$S, (\exists x)B \vdash C$ $S, B[a'/x] \vdash C$	$S \vdash (\exists x)B$ $S, \neg(\exists x)B \vdash B[a/x]$	exR* a
allL a	$S, (\forall x)B \vdash C$ $S, B[a/x] \vdash C$	$S \vdash (\forall x)B$ $S \vdash B[a'/x]$	allR

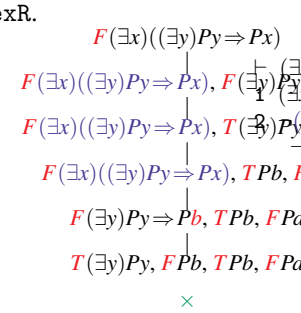
Table 9.1 Translation of tableaux rules into refinement rules

The block tableau for $\neg((\forall x)Px) \vee (Pa \wedge Pb)$ and its translation are shown below. The translation needs to preserve the disjunct eliminated by the application of orR2. We use Decide A as abbreviation for the proof fragment consisting of an application of the rule Use_decidability_of A followed immediately by orL.




```
⊢ ¬((∀x)Px)∨(Pa∧Pb)                                  by Decide (∀x)Px
1 (∀x)Px ⊢ ¬((∀x)Px)∨(Pa∧Pb)                          by orR2
1.1 (∀x)Px ⊢ Pa∧Pb                                   by andR
1.1.1 (∀x)Px ⊢ Pa                                    by allL a
1.1.1.1 (∀x)Px, Pa ⊢ Pa                              by axiom
1.1.2 (∀x)Px ⊢ Pb                                    by allL b
1.1.2.1 (∀x)Px, Pb ⊢ Pb                              by axiom
2 ¬((∀x)Px) ⊢ ¬((∀x)Px)∨(Pa∧Pb)                      by orR1
2.1 ¬((∀x)Px) ⊢ ¬((∀x)Px)                            by axiom
```

The block tableau for $(\exists x)((\exists y)Py \Rightarrow Px)$ and its translation are shown below. The translation needs to preserve the conclusion which is eliminated by the application of `exR`.




```
⊢ (∃x)((∃y)Py⇒Px)                                    by Decide (∃x)((∃y)Py⇒Px)
1 (∃x)((∃y)Py⇒Px) ⊢ (∃x)((∃y)Py⇒Px)                   by axiom
2 (∃x)((∃y)Py⇒Px) ⊢ (∃x)((∃y)Py⇒Px)                   by exR a
¬(∃x)((∃y)Py⇒Px) ⊢ (∃y)Py⇒Pa                         by impliesR
¬(∃x)((∃y)Py⇒Px), (∃y)Py ⊢ Pa                        by exL
¬(∃x)((∃y)Py⇒Px), Pb ⊢ Pa                            by notL
¬(...), Pb ⊢ (∃x)((∃y)Py⇒Px)                         by exR b
¬(...), Pb ⊢ (∃y)Py⇒Pb                               by impliesR
¬(...), Pb, (∃y)Py ⊢ Pb                              by axiom
```

9.6 Exercises

Questions

(1) Solve the first group of exercises on page 56 of Smullyan's book. Submit your solutions for the second, third, and last one.

(2) Give Refinement Logic proofs for the seventh and last of the above exercises and construct the evidence terms from the proofs.
Show by giving counterexamples why the converse is not valid.

(3) Solve the second group of exercises on page 56 of Smullyan's book. Submit your solutions for the sixth and the last one.

(4) Solve the exercise on page 63 of Smullyan's book.
(In contrast to Homework 5 (4) you now have to prove *that the given formula is not finitely satisfiable)*

Chapter 10
Correctness, Completeness, and Compactness

10.1 Overview

Now that we have introduced a proof calculus for first-order logic we have to address the usual questions again, that always come up when dealing with formal proof systems.

(1) *Is the tableau method correct? Can we be sure that a proven formula is in fact valid?*
(2) *Is it complete? Can we prove every valid formula with the tableau method?*
(3) *Is it decidable? Does it always tell us whether a formula is valid or not?*
(4) *What about compactness? What does the satisfiability of finite sets of formulas tell us?*
(5) *Are there proof strategies for building first-order tableaux that are more successful or more efficient than others?*

10.2 Correctness of First-Order Tableaux

To prove the correctness of the tableau method, one has to show that *the origin of a closed tableau is unsatisfiable* or, equivalently, that a tableau is satisfiable and cannot be closed whenever the formula at its origin is satisfiable. The basic structure of the proof is the same as the one for propositional logic, so we just formulate the key insights here.

Let $\mathscr{U}$ be an arbitrary universe and v be a first-order valuation of $E^{\mathscr{U}}$ (φ is the identity mapping).

F_1: α is true under v, if and only if α_1 and α_2 are true under v
F_2: β is true under v, if and only if at least one of β_1 and β_2 is true under v
F_3: γ is true under v, if and only if $\gamma(k)$ is true under v for every $k \in \mathscr{U}$
F_4: δ is true under v, if and only if $\delta(k)$ is true under v for at least one $k \in \mathscr{U}$

These facts follow immediately from the definition of first-order valuations on $E^{\mathscr{U}}$. As a consequence we can show the following laws about the satisfiability of sets of formulas with parameters.

Let S be any set of formulas

G_1: If S is satisfiable and $\alpha \in S$, then $S \cup \{\alpha_1, \alpha_2\}$ is satisfiable
G_2: If S is satisfiable and $\beta \in S$, then at least one of $S \cup \{\beta_1\}$ and $S \cup \{\beta_2\}$ is satisfiable
G_3: If S is satisfiable and $\gamma \in S$, then $S \cup \{\gamma(a)\}$ is satisfiable for every parameter a
G_4: If S is satisfiable and $\delta \in S$, then $S \cup \{\delta(a)\}$ is satisfiable for every parameter a that does not occur in S

The first three laws are obvious but the last one is not, as it shows how to represent the semantical "for at least one $k \in \mathscr{U}$" by a syntactical requirement "for every new parameter a".

Proof. Let $\mathscr{U}$ be a universe, v be a first-order valuation, and φ be a mapping from the set of parameters to $\mathscr{U}$ such that all $A \in S$ are true under φ and v. Since $\delta \in S$, there must be at least one parameter a such that $\delta(a)$ is true under φ and v.[1]

Let $k = \varphi(a) \in \mathscr{U}$ and define $\varphi'(b) = \begin{cases} \varphi(b) & \text{if } b \text{ occurs in } S \\ k & \text{otherwise} \end{cases}$.

Then all $A \in S$ are true under φ' and v (nothing has changed) and for every parameter a' that does not occur in S we have $\varphi'(a') = \varphi(a) = k$ and thus $\mathrm{Em}v[\delta(a')^{\varphi'}] = v[\delta(a)^{\varphi}] = t$. Hence $S \cup \{\delta(a')\}$ is satisfied by $(\mathscr{U}, v\varphi')$

The remainder of the correctness proof is almost identical to the propositional case. We have to prove that every tableau with a satisfiable origin contains at least one satisfiable path.

Theorem 10.1. *Let $\mathscr{T}$ be an arbitrary tableaux whose root is satisfiable. Then there is a path ϑ in $\mathscr{T}$ that is uniformly satisfiable.*

Proof. We use structural induction on tableau trees.

base case: If $\mathscr{T}$ has just a single point, then let ϑ be the path consisting of the root of $\mathscr{T}$.

step case: Assume the statement holds for some $\mathscr{T}$. Let $\mathscr{T}_1$ be a direct extension of $\mathscr{T}$ and I be a model for the root of $\mathscr{T}_1$. Since $\mathscr{T}$ and $\mathscr{T}_1$ have the same root there is a satisfiable path ϑ in $\mathscr{T}$.

We consider 5 cases (the first 3 are identical to what we had before)

(1) If $\mathscr{T}_1$ does *not* extend $\mathscr{T}$ at ϑ, then ϑ is a satisfiable path ϑ in $\mathscr{T}_1$.
(2) If $\mathscr{T}_1$ extends $\mathscr{T}$ at ϑ by some α_i, then we know that α is on ϑ. Thus $\vartheta_1 = \vartheta \circ \alpha_i$ is a satisfiable path in $\mathscr{T}_1$ by G_1.
(3) If $\mathscr{T}_1$ extends $\mathscr{T}$ at ϑ by β_1 and β_2 then β is on ϑ and $\vartheta_1 = \vartheta \circ \beta_1$ or $\vartheta_1 = \vartheta \circ \beta_2$ is a satisfiable path in $\mathscr{T}_1$ by G_2.
(4) If $\mathscr{T}_1$ extends $\mathscr{T}$ at ϑ by some $\gamma(a)$, then γ is on ϑ and $\vartheta_1 = \vartheta \circ \gamma(a)$ is a satisfiable path in $\mathscr{T}_1$ by G_3.

[1] This is not entirely true if there are elements u in the universe that have no parameter a with $\varphi(a) = u$. In that case the proof argument needs a few more details.

(5) If $\mathscr{T}_1$ extends $\mathscr{T}$ at ϑ by some $\delta(a)$ then δ is on ϑ and a does not occur in any of the formulas of ϑ. Thus $\vartheta_1 = \vartheta \circ \delta(a)$ is a satisfiable path in $\mathscr{T}_1$ by G_4.

As a consequence, every closed tableau has an unsatisfiable root, which means that the (unsigned) formula at the root of the tableau must be valid.

10.3 Completeness

Proving the completeness of a first-order calculus gives us Gödel's famous completeness result. Gödel proved it for a slightly different proof calculus, and the proof that we will show here goes back to Beth and Hintikka. Let us briefly resume the propositional case.

The key to the completeness proof was the use of Hintikka's lemma, which states that every downward saturated set, finite or not, is satisfiable. We then showed that every open and complete path is in fact a Hintikka sequence. Putting these two things together we reasoned that the root of an open and complete tableau must be satisfiable. Thus a complete tableau for a valid formula cannot be open which means that every tableau for a valid formula will eventually close.

We will prove the first order case along these lines, but have to keep in mind that several things have changed.

- The definition of a valuation now includes quantifiers.
- The definition of Hintikka sets must take γ and δ formulas into account.
- The notion of a complete tableau needs to be adjusted, because there is now the possibility of non-terminating proof attempts.

Fortunately, we can easily make the necessary adjustments and then proceed as before. First, let us define first-order Hintikka sets. A *Hintikka Set for a universe U* is a set S of U-formulas such that for all closed U-formulas A, α, β, γ, and δ the following conditions hold.

H_0: If A is atomic and $A \in S$ then $\bar{A} \notin S$
H_1: If $\alpha \in S$ then $\alpha_1 \in S \wedge \alpha_2 \in S$
H_2: If $\beta \in S$ then $\beta_1 \in S \vee \beta_2 \in S$
H_3: If $\gamma \in S$ then $\forall k \in U.\ \gamma(k) \in S$
H_4: If $\delta \in S$ then $\exists k \in U.\ \delta(k) \in S$

The first axiom expresses the openness of S while the other four state that it is downward saturated. Note that because of axiom H_3, Hintikka sets are usually infinite, unless the universe is finite. But the proof of Hintikka's lemma that we discussed a few weeks ago did not depend on the fact that the set is finite, so it can easily be adapted to the first-order case.

Theorem 10.2 (Hintikka Lemma). *Every Hintikka set is uniformly satisfiable*

Proof. Because of axiom H_0 we can define a valuation that satisfies all the atomic formulas in S.

Define $\mathsf{v}(P(k_1,\ldots,k_n)) = \begin{cases} \mathtt{f} \text{ if } FP(k_1,\ldots,k_n) \in S \\ \mathtt{t} \text{ otherwise} \end{cases}$

To show that v satisfies every formula $Y \in S$ we proceed by structural induction on formulas, keeping in mind that the cases for γ and δ are straightforward generalizations of those for α and β.

base case: If $Y \in S$ is an atomic formula then $v[Y]=t$ by definition.

step case: Assume the the claim holds for all subformulas of Y.

- If Y is of type α then $\alpha_1,\alpha_2 \in S$, hence by assumption $v[\alpha_1]=v[\alpha_2]=t$. With the definition of first-order valuations we get $v[Y]=t$.
- If Y is of type β then $\beta_1 \in S$ or $\beta_2 \in S$, hence $v[\beta_1]=t$ or $v[\beta_2]=t$ and thus $v[Y]=t$.
- If Y is of type γ then $\gamma(k) \in S$ and hence $v[\gamma(k)]=t$ for all $k \in U$, thus $v[Y]=t$.
- If Y is of type δ then $\delta(k) \in S$ and hence $v[\delta(k)]=t$ for some $k \in U$, thus $v[Y]=t$.

Now what about the completeness of a tableau? In the propositional case, this meant that the tableau cannot be extended any further, because all formulas have been decomposed. Since the propositional tableau method terminates after finitely many steps, this was easy to define. In the first-order case, however, we have to be a bit more careful.

We know that because of γ-formulas proofs may have infinite branches. But that is not the main problem, since Hintikka's lemma also works for infinite sets. However, not every infinite branch in a tableau is automatically a Hintikka set.

Consider for example, the formula `∃x,y.P(x,y)`, which is certainly not valid. Thus `F∃x,y.P(x,y)` is satisfiable and because of the correctness of the tableau method we know that every proof attempt will fail. But does *every* failing proof attempt actually give us the Hintikka set that we need to reason that `F∃x,y.P(x,y)` *must* be satisfiable?

Certainy not. Just imagine we start decomposing the main formula, which is a γ formula, over and over again. Then we can go on and on forever without ever touching the inner γ formula and we get an infinite branch that does not satisfy the third Hintikka axiom for this inner γ formula.

So our completeness proof cannot rely on an arbitrary attempt to find a tableau proof. After all, completeness only says that *it must be possible to prove every valid formula correct with the tableau method* but it doesn't require that *any* attempt will succeed. And the fact that we weren't able to find a proof with a not so bright approach doesn't mean that there is none at all.

Fortunately, we can design a systematic approach that is guaranteed to find a tableau proof, provided there is one. And we will show that using this systematic method we will find a tableau proof for every valid formula.

Essentially, a systematic method only has to describe a treatment of γ formulas that guarantees axiom H_3. The α, β, and δ rules make sure that the other Hintikka axioms are always satisfied.

How can we make sure that all γ formulas are eventually covered completely?

We have to proceed similarly to an enumeration of lists of integers. We modify the extension procedure for tableaux in a way that each γ formula, and thus every other formula as well, will be revisited on a regular basis.

A systematic procedure for proving a first-order formula X

Start with the signed formula FX and recursively extend the tableau as follows:

- *If the tableau is already closed then stop. The formula is valid.*
- *Otherwise select a node Y in the tableau that is of* minimal level *wrt. the still unused nodes and extend* every *open branch ϑ through Y as follows:*
  - *If Y is α extend ϑ to $\vartheta \cup \{\alpha_1, \alpha_2\}$.*
  - *If Y is β, extend ϑ to two branches $\vartheta \cup \{\beta_1\}$ and $\vartheta \cup \{\beta_2\}$.*
  - *If Y is γ, extend ϑ to $\vartheta \cup \{\gamma(a), \gamma\}$, where a is the first parameter that is not on ϑ.*
  - *If Y is δ, extend ϑ to $\vartheta \cup \{\delta(a)\}$, where a is the first parameter that does not occur in the tableau tree.*

Thus the procedure always copies a γ formula to the end of a branch when it is being considered. This way we make sure that it is considered over and over again, but that all the other formulas on the branch are decomposed before that. Thus in the end all the formulas are being used, because we have only denumerably many parameters. This method is not very efficient, but it works.

Using the systematic procedure we can give a new definition of complete tableau. A systematic tableau is called *finished* if it is either infinite or finite and cannot be extended any further. With this definition we immediately get the following result.

Lemma 10.3. *In every finished systematic tableau, every open branch is a Hintikka sequence.*

A detailed proof for this lemma would show by structural induction that the systematic method does in fact cover all the required formulas. Together with Hintikka's lemma we get.

Corollary 10.4. *In every finished systematic tableau, every open branch is uniformly satisfiable.*

As before, the completeness theorem is now an immediate consequence.

Theorem 10.5 (Completeness theorem for first-order logic).
If a first-order formula X is valid, then X is provable. Furthermore the systematic tableau method will construct a closed tableau for FX after finitely many steps.

The first statement follows from the above corollary by contraposition and the fact that the systematic tableau method always "constructs" a finished tableau. As for the second, a closed tableau can only have finite branches, which – according to König's lemma – means that it must be finite.

Note that correctness and completeness is preserved again if we require an *atomically closed tableau*, i.e. a tableau where branches only close if there is an atomc formula and its conjugate. Correctness follows from the fact that an atomically closed tableau is certainly a closed tableau, while the systematic tableau method makes sure that we construct a Hintikka sequence if the tableau does not close (which is the case if it does not close atomically). Hintikka's lemma thus implies

Corollary 10.6.
If a first-order formula X is valid, then X the there is an atomically closed tableau for FX.

The corollary, together with the systematic method, also has another important consequence that will be relevant for the compactness of first-order logic.

Theorem 10.7 (Löwenheim theorem for first-order logic).
If a first-order formula X is satisfiable, then it is satisfiable in a denumerable domain.

The proof for this theorem is based on the observation that the systematic tableau method uses only denumerably many parameters to build a Hintikka sequence if the tableau doesn't close. Since a tableau with a satisfiable formula at its root cannot close, it must contain an open branch ϑ with at most denumerably many parameters. As this branch is uniformly satisfiable it satisfies X in a denumerable domain (the subset of the domain U that represents the set of parameters on ϑ).

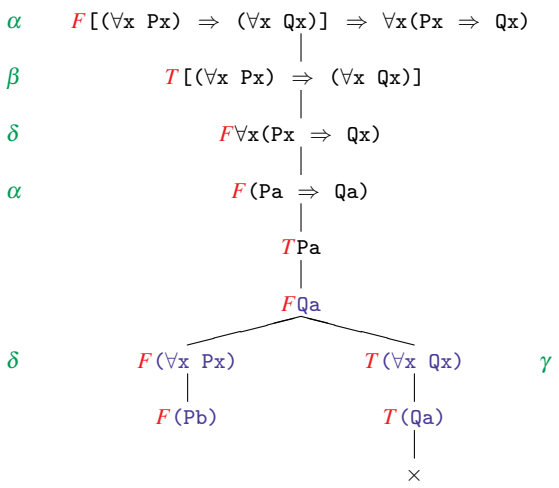
Thus if the formula is intended to describe properties of, for instance, the real numbers there will be a denumerable subset of the real numbers in which the property is already satisfied.

10.4 Decidability Issues

While the tableau method could be used as decision procedure for propositional logic, this will not work in first-order logic anymore. If a formula is not valid, the systematic method may lead to an infinite tableau. This is, however, not a deficiency of the tableau method. In fact, there is no correct and complete proof method for first-order logic that always terminates, as first-order logic is known to undecidable.

Nevertheless, in some cases the tableau method can decide that a formula is invalid although the proof is not finished yet. Whenever we have constructed a branch ϑ that represents a Hintkka set (over the finite domain of the parameters that occur on ϑ), then we know that the origin FX of the tableau is satisfiable and hence X must be invalid. In these rare cases, the Hintikka branch gives us a *counterexample* for the validity of the formula.

Example: Consider the invalid formula `[(∀x Px) ⇒ (∀x Qx)] ⇒ ∀x(Px ⇒ Qx)`.



The tableau to the left cannot be extended anymore in any meaningful way and has one open branch ϑ, which is a Hintikka set for the 2 element domain $\mathsf{U} = \{\mathtt{a},\mathtt{b}\}$. In this branch the marked α, β, γ, and δ points are fulfilled for the domain U, since all formulas are true under the atomic valuation that assigns `f` to `Qa` and `Pb` and `t` to `Pa`.
In particular, the original formula `[(∀x Px) ⇒ (∀x Qx)] ⇒ ∀x(Px ⇒ Qx)` evaluates to `f` under this interpretation.

It is possible to build this "Hintikka Test" into the tableau method and use it to prove that certain formulas cannot be valid. However, there are many formulas that are neither valid nor falsifiable in any finite domain. Any tableau proof attempt for these will run infinitely and at no stage of the proof will we know whether the formula is valid or not.

There are also formulas that are falsifiable, but not in any finite domain. However, the tableau method is "finite" and therefore not suited to produce counterexamples for them.

10.5 Compactness

The final important property of first-order logic that we have to investigate is compactness: *Given a set F of first-order formulas, what does the satisfiability of finite subsets tell us about the satisfiability of the whole set*. In propositional logic we have shown that a set S is uniformly satisfiable if all of its finite subsets are. We gave three proofs for that: one using tableau proofs and König's lemma, one giving a direct construction of a Hintikka set, and one using Lindenbaum's construction, extending S to a maximally consistent set, which turned out to be a proof set.

In first-order logic the question of compactness leads to a spin-off question. Can we extend Löwenheim's theorem to sets of formulas and prove "*if a set of formulas is satisfiable then it is satisfiable in a denumerable domain*"? This is the so-called

Skolem-Löwenheim theorem and it follows from an extended form of compactness, which gives us uniform satisfiability over a denumerable domain if we know that all finite subsets are satisfiable? In the following we will use the tableau method to prove these facts.

Recall that in the propositional case we proved compactness by systematically constructing an tableau for the set S, using the fact that every finite subset of S is satisfiable to ensure that the tableau is infinite. We then used König's lemma to show that this tableau has an infinite branch, which in turn must be a Hintikka set. The construction of the tableau made sure that S is a subset of that set and hence satisfiable.

We will proceed in a similar fashoion for first-order logic. First, we define a *first-order tableau for a set S of pure formulas* (i.e formulas without parameters). Such a tableau starts with an arbitrary element of S at its origin and is then constructed by applying either one of the four rules α, , γ, or δ, or by adding another element of S to the end of an open branch. The elements of S so added are called the *premises* of the tableau. We call a tableau *complete* if every open branch is a Hintikka set for the universe of parameters and contains all the elements of S. Obviously every closed tableau is complete as well.
We first show that a complete tableau can be constructed for every set of first-order formulas.

Lemma 10.8. *For every denumerable set S there is a complete tableau for S.*

Proof. We construct the desired tableau by combining our systematic proof procedure with the construction of a tableau for S that we used in the propositional case. Arrange S as a denumerable sequence $X_1, X_2, X_3, \ldots, x_n, \ldots$.

We begin by placing X_1 at the origin of the tableau. This concludes stage 1. In stage $n+1$ we extend the tableau constructed at stage n as follows.

- If the tableau is already closed then stop. The formula is valid.
- Otherwise select a node Y in the tableau that is of *minimal level* wrt. the still unused nodes, extend *every* open branch ϑ through Y as in the systematic procedure, and add X_{n+1} to the end of every open branch.

By construction every open branch in the resulting tableau is a Hintikka set for the universe of parameters (we used the systematic method) and contains the set S.

Using the method for constructing complete tableaux we can find an unsatisfiable finite subset for every set of formula that has a closed tableau.

Lemma 10.9. *If a pure set S has a closed tableau, then a finite subset of S is unsatisfiable.*

Proof. Assume S has a closed tableau $\mathscr{T}$ and consider the set S_p of premises of $\mathscr{T}$. By König's lemma, $\mathscr{T}$ must be finite and so is S_p. S_p must be unsatisfiable, since otherwise every branch containing S_p would be open (recall that by construction the elements of the branch are derived from the formulas in S_p using tableau rules only.)

Now we use contraposition to prove "denumerable compactness.

Theorem 10.10.
If all finite subsets of a denumerable set S of pure formulas are satisfiable, then S is uniformly satisfiable in a denumerable domain.

Proof. Let $\mathscr{T}$ be a complete tableau for S. Since all finite subsets of S are satisfiable, $\mathscr{T}$ cannot be closed due to the above lemma, so it has an open branch ϑ. Since $\mathscr{T}$ is complete, ϑ is a Hintikka set for the denumerable universe of parameters contained in S. Thus S is uniformly satisfiable in a denumerable universe.

Theorem 10.10 gives us the two desired results and another insight as immediate consequences.

Corollary 10.11 (Compactness of First-Order Logic).
If all finite subsets of a pure set S are satisfiable, then S is uniformly satisfiable

Corollary 10.12 (Skolem-Löwenheim theorem for First-Order Logic).
If a pure set S of is satisfiable then it is satisfiable in a denumerable domain.

Corollary 10.13.
If no tableau for a pure set S can close, then S is satisfiable in a denumerable domain.

The last corollary leads to a lot of interesting results about theoretical properties of first-order logic that we won't discuss in this course. Those of you who are interested may study Smullyan's chapters VI and VII.