Formalizing Basic First Order Model Theory

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA
johnh@ichips.intel.com

Abstract. We define the syntax of unsorted first order logic as a HOL datatype and define the semantics of terms and formulas, and hence notions such as validity and satisfiability. We prove formally in HOL some elementary metatheorems such as Compactness, Löwenheim-Skolem and Uniformity, via canonical term models. The proofs are based on those in Kreisel and Krivine's book on model theory, but the HOL formalization raises several interesting issues. Because of the limited nature of type quantification in HOL, many of the theorems are awkward to state or prove in their standard form. Moreover, simple and elegant though the proofs seem, there are surprising difficulties formalizing Skolemization, one of the more intuitively obvious parts. On the other hand, we significantly improve on the original textbook versions of the arguments, proving two of the main theorems together rather than by separate arguments.

1 Introduction

This paper deals with the formalization, in the HOL Light theorem prover, of the basic model theory of first order logic. Our original motivation was that we were writing a textbook on mathematical logic, and wanted to make sure that when presenting some of the major results, we didn't make any slips – hence the desire for machine checking. We took as our model for this part of logic the textbook of Kreisel and Krivine [6], hereinafter referred to as K&K. However, as we shall see, in the process of formalizing the proofs we made some significant improvements to their presentation.

In view of the considerable amount of mathematics that has been formalized, mainly in Mizar [14, 11], it is perhaps hardly noteworthy to formalize yet another fragment. However, we believe that the present work does at least raise a few interesting general points.

- Formalization of syntax constructions involving bound variables has inspired a slew of research; see e.g. Chap. 3 of Pollack [9], or Gordon and Melham [3]. We offer an unusual angle, in that for us, syntax is subordinate to semantics.
- The apparent intuitive difficulty of parts of the proof does not correlate well with the difficulty of the HOL formalization: an apparently straightforward part turns out to be the most difficult. This raises interesting questions over whether the textbook or HOL is at fault.

– As part of the task of formalization, we found improvements to the textbook proof. Though this was an indirect consequence of formalization, resulting from the necessary close reading of the text, it serves as a good example of possible auxiliary benefits.
– We provide a good illustration of how HOL's simplistic type system can be a hindrance in stating results in the most natural way. In particular, care is needed because quantification of type variables happens implicitly at the sequent level, rather than via explicit binding constructs.

HOL Light is our own version of HOL, and while its logical axiomatization, proof tools and even implementation language are somewhat different from other versions, the underlying logic is exactly equivalent to the one described by Gordon and Melham [4]. This is a version of simply typed lambda calculus used as a foundation for classical higher order logic.

2 Syntactic Definitions

We define first order terms in HOL as a free recursive type: a term is either a variable, identified by a numeric code, or else a function symbol, also identified by a number, followed by a possibly empty list of arguments. (We regard nullary functions as individual constants.)

```
term = V num
     | Fn num (term list)
```

Here we have already made two significant choices. First of all, we have restricted ourselves to countable languages, since the function symbols are indexed by $\mathbb{N}$. This restriction is inessential, however; everything that follows extends to any infinite HOL type, given a theorem that for an infinite type α there is an injection $\alpha \times \alpha \to \alpha$, a fairly easy consequence of Zorn's Lemma. The restriction to countable languages was made entirely for convenience, to avoid having to specify types everywhere.

Secondly, we have not made any restrictions on the arities of functions, e.g. that function 6 can only be applied to two arguments. Rather, we really consider a function symbol as identified by a pair consisting of its numerical code and its arity, so in $f_3(x)$ and $f_3(x, y)$, the two functions are different, identified by the pairs $(3, 1)$ and $(3, 2)$ respectively. In fact, we define a function that returns the set of functions, in this sense, that occur in a term:

```
|- (∀v. functions_term (V v) = {}) ∧
   (∀f l. functions_term (Fn f l) =
        (f,LENGTH l) INSERT (LIST_UNION (MAP functions_term l)))
```

where:

```
|- (LIST_UNION [] = {}) ∧
   (LIST_UNION (CONS h t) = h UNION (LIST_UNION t))
```

Next we define formulas; there is a wide choice over which connectives to take as primitive and which as defined. We differ a bit from our textbook model by taking falsity, atoms, implications and universal quantifications as primitive.

```
form = False
     | Atom num (term list)
     | --> form form
     | !! num form
```

Once again, identically-numbered predicates used with different arities are considered different. We now define functions to return the set of functions and predicates in this sense occurring in a formula:

```
|- (functions_form False = {}) ∧
   (functions_form (Atom a l) = LIST_UNION (MAP functions_term l)) ∧
   (functions_form (p --> q) =
        (functions_form p) UNION (functions_form q)) ∧
   (functions_form (!! x p) = functions_form p)

|- (predicates_form False = {}) ∧
   (predicates_form (Atom a l) = {(a,LENGTH l)}) ∧
   (predicates_form (p --> q) =
        (predicates_form p) UNION (predicates_form q)) ∧
   (predicates_form (!! x p) = predicates_form p)
```

It's more convenient to lift these up to the level of sets of formulas, and we pair up the sets of functions and predicates as the so-called *language* of a set of formulas.

```
|- functions fms = UNIONS {functions_form f | f IN fms}

|- predicates fms = UNIONS {predicates_form f | f IN fms}

|- language fms = functions fms, predicates fms
```

Trivially we have then, for a singleton set of formulas:

```
|- language {p} = functions_form p,predicates_form p
```

Other logical constants are defined in a fairly standard way; it doesn't really matter for what follows exactly how this is done. These are respectively negation, truth, disjunction (or), conjunction (and), bi-implication (iff) and existential quantification.

```
|- Not p = p --> False

|- True = Not False

|- p || q = (p --> q) --> q

|- p && q = Not (Not p || Not q)

|- p <-> q = (p --> q) && (q --> p)

|- ?? x p = Not(!!x (Not p))
```

The set of free variables in a term and formula are now defined:

```
|- (∀x. FVT (V x) = {x}) ∧
   (∀f l. FVT (Fn f l) = LIST_UNION (MAP FVT l))

|- (FV False = {}) ∧
   (∀a l. FV (Atom a l) = LIST_UNION (MAP FVT l)) ∧
   (∀p q. FV (p --> q) = FV p UNION FV q) ∧
   (∀x p. FV (!! x p) = FV p DELETE x)
```

We prove various simple consequences of the definition, most importantly, by induction over the syntax of terms and formulas, that these always give finite sets, e.g.

```
|- ∀p. FINITE(FV p)
```

The most complex syntactic definition is of substitution. We have chosen a 'name-carrying' formalization of syntax, rather than indexing bound variables using some scheme following de Bruijn [1]. The latter is usually preferred when formalizing logical syntax precisely because substitution is simpler to define correctly. (Indeed, it is foreshadowed in the tradition – see Prawitz [10] for example – of distinguishing between [bound] variables and [free] parameters.) Our decision to stick with name-carrying terms was partly in order to stay closer to intuition, and as we shall see, defining substitution so that it renames variables appropriately is not too difficult. In some sense, things are easier for us since we always have semantics (to be defined later) as a check. By contrast, in formalizing, say, beta reduction in lambda-calculus, the syntax is the primary object of investigation and substitution must be defined in a way that is correct and intuitively clear or the whole exercise loses its point.

In order to define renaming substitution as a clean structural recursion over terms, it's necessary to define multiple substitutions rather than single ones [13]. This is because for something like $(\forall x.\, P(x) \Rightarrow P(y))[x/y]$ we need recursively to perform two substitutions on the body of the quantifier, one to substitute for y and one to rename the variable x. Therefore we take multiple parallel substitution as primitive. (Multiple serial substitution would also be acceptable, since by

construction the substitutions in renaming clauses never interfere, but this seems less intuitive.) We represent substitutions simply as functions `:num->term`, mapping each variable index to the term to be substituted for that variable. (The identity substitution is therefore the type constructor for variable terms, `V`.) Although in practice we only need substitutions that are nontrivial (i.e. differ from `V`) only for finitely many arguments, nothing seems to be gained by imposing this restriction generally. Substitution at the term level is simple:

```
|- (∀x. termsubst v (V x) = v(x)) ∧
   (∀f l. termsubst v (Fn f l) = Fn f (MAP (termsubst v) l))
```

It's when we come to formulas that the complexities of renaming arise. We need to see if a variable in the substituted body would become captured by the bound variable in a universal quantifier, and if so, rename the bound variable appropriately.

```
|- (formsubst v False = False) ∧
   (formsubst v (Atom p l) = Atom p (MAP (termsubst v) l)) ∧
   (formsubst v (q --> r) = (formsubst v q --> formsubst v r)) ∧
   (formsubst v (!!x q) =
        let v' = valmod (x,V x) v in
        let z = if ∃y. y IN FV(!!x q) ∧ x IN FVT(v'(y))
                then VARIANT(FV(formsubst v' q)) else x in
        !!z (formsubst (valmod (x,V(z)) v) q))
```

Here the function `valmod` modifies a substitution, or more generally any function, as follows:

```
|- valmod (x,a) v = λy. if y = x then a else v y
```

while `VARIANT` picks a variable not in a given set. It is defined constructively over finite sets to find the maximum free variable in the set and add one to it. An important theorem gives the free variables in a substituted formula.

```
|- FV(formsubst i p) = {x | ∃y. y IN FV(p) ∧ x IN FVT(i y)}
```

while another states that only variables free in the formula are relevant to the outcome of performing a substitution.

```
|- (∀x. x IN (FV p) ⇒ (v'(x) = v(x)))
   ⇒ (formsubst v' p = formsubst v p)
```

3 Semantic Definitions

The above syntactic definitions are really only a tool; we are primarily interested in semantics. The first step is to define the notion of an interpretation in HOL:

it is simply a triple consisting of a domain, interpretations for the function symbols and interpretations for the relation symbols. The three parts of a triple are selected by the HOL functions `Dom`, `Fun` and `Pred` respectively. The truth or falsity of a formula, or more generally the element of an interpretation's domain selected by a term, depends in general not just on the interpretation `M` but also on a valuation of free variables `v`. Only at the top level, so to speak, if the formula has no free variables, does the valuation cease to play a role. Accordingly the valuations of terms and formulas are defined w.r.t. an interpretation and a valuation:

```
|- (∀x. termval M v (V x) = v(x)) ∧
   (∀f l. termval M v (Fn f l) = Fun(M) f (MAP (termval M v) l))

|- (holds M v False = F) ∧
   (∀a l. holds M v (Atom a l) = Pred(M) a (MAP (termval M v) l)) ∧
   (∀p q. holds M v (p --> q) = holds M v p ⇒ holds M v q) ∧
   (∀x p. holds M v (!! x p) =
               ∀a. a IN Dom(M) ⇒ holds M (valmod (x,a) v) p)
```

Once again, whether a formula holds depends only on the valuation of the free variables:

```
|- (∀x. x IN (FV p) ⇒ (v'(x) = v(x))) ⇒ (holds M v' p = holds M v p)
```

and so in particular for a formula with no free variables, `v` plays no role. Analogously, the interpretation of function symbols given by the interpretation is only important for the language of the formula concerned:

```
|- (Dom(M) = Dom(M')) ∧
   (∀P zs. Pred(M) P zs = Pred(M') P zs) ∧
   (∀f zs. (f,LENGTH zs) IN functions_form p
           ⇒ (Fun(M) f zs = Fun(M') f zs))
   ⇒ ∀v. holds M v p = holds M' v p
```

One of the main theorems relates substitution and holding in an interpretation. The proof is quite straightforward, though involves a slightly messy case analysis. Its intuitive plausibility is good evidence that our definition of substitution is satisfactory, and, indeed, its later usefulness means that from now on we can largely forget the details of how substitution was defined.

```
|- holds M v (formsubst i p) = holds M (termval M v o i) p
```

We have not yet imposed any restrictions on interpretations. For some purposes, like the above theorems, they are not needed, and it seems appropriate not to restrict their generality. But we often need to assume that the domain of an interpretation is nonempty; this is generally done in the logical literature, though a few authors like Johnstone [5] try to avoid it. More importantly, the

interpretations of functions must indeed be functions back into the appropriate domain. We abbreviate this by saying that a triple is an interpretation of a particular language, and define it in HOL as follows:

```
|- interpretation (fns,preds) M =
     ∀f l. (f,LENGTH l) IN fns ∧ FORALL (λx. x IN Dom(M)) l
           ⇒ (Fun(M) f l) IN Dom(M)
```

Similarly, we are only really interested in valuations that map into the domain of an interpretation, and so we define:

```
|- valuation(M) v = ∀x. v(x) IN Dom(M)
```

We also define what it means for an interpretation to satisfy (be a model of) a set of formulas:

```
|- M satisfies fms = ∀v p. valuation(M) v ∧ p IN fms ⇒ holds M v p
```

We might wish to define satisfiability as:

```
satisfiable fms = ∃M. M satisfies fms
```

However such a definition cannot be made with the appropriate type variable on the right-hand side; one can only define a notion of being satisfiable in models of a certain type. This is ultimately unimportant because we will prove in what follows that a formula is satisfiable iff it is satisfiable over the natural numbers (the Löwenheim-Skolem theorem). In any case, even in systems like ZF set theory which allow us to state the required quantification directly, we must in general take care that it corresponds to the intuitive notion. For second order logic, the identification of validity in a ZF-like cumulative set theory with a more general notion ('Kreisel's principle') is known to be a strong assumption, entailing higher order reflection principles [12].

4 Proofs of the Metatheorems

The proofs of the metatheorems given by K&K are based on several steps:

- Compactness for the propositional subsystem
- Prenex normal form and Skolem normal form.
- Use of canonical models

We will consider each of these steps separately.

4.1 Propositional Logic

We define the class of quantifier-free formulas by recursion over the structure of formulas.

```
|- (qfree False = T) ∧
   (qfree (Atom n l) = T) ∧
   (qfree (p --> q) = qfree p ∧ qfree q) ∧
   (qfree (!!x p) = F)
```

If a formula is quantifier-free, we can look at it not as a first order formula based on the appropriate language, but rather as a formula of propositional logic based on atomic first order formulas. In this case, we have a separate notion of holding in a valuation, which in this context is a function of type `term->bool` indicating whether each atom is considered true or false.

```
|- (pholds v False = F) ∧
   (pholds v (Atom p l) = v (Atom p l)) ∧
   (pholds v (q --> r) = pholds v q ⇒ pholds v r) ∧
   (pholds v (!!x q) = v (!!x q))
```

The last clause turns out to be irrelevant to the later proofs, since the function `pholds` is only used for quantifier-free formulas. We chose to define it this way, treating quantified formulas as atoms, with the vague idea of later formalizing a first order proof system given by Enderton [2], which has the nice feature that a formula is valid iff it follows propositionally from the (infinite) set of axioms, interpreting quantified formulas as atomic. However at time of writing we haven't tackled this.

The key theorem for the propositional subsystem is compactness, which states that if all finite subsets of a set of (quantifier-free) formulas are propositionally satisfiable, then so is the set as a whole.

```
|- (∀p. p IN A ⇒ qfree p) ∧
   (∀B. FINITE(B) ∧ B SUBSET A ⇒ ∃d. ∀r. r IN B ⇒ pholds(d) r)
   ⇒ ∃d. ∀r. r IN A ⇒ pholds(d) r
```

The proof is a fairly routine application of Zorn's Lemma, already proved in HOL from the primitive form of the Axiom of Choice. We start by defining the notions of propositional satisfiability and finite satisfiability:

```
|- psatisfiable s = ∃v. ∀p. p IN s ⇒ pholds v p

|- finsat s = ∀t. t SUBSET s ∧ FINITE(t) ⇒ psatisfiable t
```

We can now rephrase compactness as: if a set is finitely propositionally satisfiable, and contains only quantifier-free formulas, then it is propositionally satisfiable. We use Zorn's Lemma to show that every finitely satisfiable set can be extended to a maximal one:

```
|- finsat(A) ⇒ ∃B. A SUBSET B ∧ finsat(B) ∧
                  ∀C. B SUBSET C ∧ finsat(C) ⇒ (C = B)
```

The use of Zorn's Lemma is unnecessary here; since the set of primitive propositions is countable, we can construct the maximal set by recursion: just enumerate the formulas (ψ_n) and then add either ψ_n or $\neg\psi_n$ to the starting set at each stage; it's easy to see this preserves finite satisfiability. However we chose to use a proof that would also work if we lifted the restriction to countable languages. K&K assert that this sort of step-by-step proof also works for any wellordered set of primitive propositions, but it requires nontrivial changes to work over a non-discrete order.

Now we can define a valuation based on the maximal finitely satisfiable set. This works because the set has the following closure properties:

```
|- finsat(B) ∧ (∀C. B SUBSET C ∧ finsat(C) ⇒ (C = B))
   ⇒ ¬(False IN B)

|- finsat(B) ∧ (∀C. B SUBSET C ∧ finsat(C) ⇒ (C = B))
   ⇒ ∀p q. (p --> q) IN B = p IN B ⇒ q IN B
```

Because of the HOL identification of sets and predicates, the maximal finitely consistent set is a satisfying valuation for itself, and *a fortiori*, for the starting set. Hence the compactness theorem is established.

4.2 Normal Forms

The next stage is to show that each formula has a prenex normal form and a (purely universal) Skolem normal form. We first define, inductively, the appropriate syntactic classes:

```
|- (∀p. qfree p ⇒ prenex p) ∧
   (∀x p. prenex p ⇒ prenex (!!x p)) ∧
   (∀x p. prenex p ⇒ prenex (??x p))

|- (∀p. qfree p ⇒ universal p) ∧
   (∀x p. universal p ⇒ universal (!!x p))
```

The goal is to prove that every formula has a logically equivalent prenex form, and also a purely universal Skolem normal form that, while not logically equivalent, is satisfiable iff the original formula is. We define constructive procedures for calculating these forms, that can be executed by conditional rewriting. This is not so much because such concreteness seems useful – though it can hardly be bad – but because otherwise we get tripped up by type quantification. Logical equivalence involves a quantification over all models with arbitrary domain types, so if we express these theorems as existence assertions:

$$\forall p.\ \exists q.\ \forall M : (\alpha)interpretation.\ E[M, p, q]$$

we have the problem that we have really stated:

$$\forall\alpha.\, \forall p.\, \exists q.\, \forall M : (\alpha)interpretation.\, E[M, p, q]$$

rather than

$$\forall p.\, \exists q.\, \forall\alpha.\, \forall M : (\alpha)interpretation.\, E[M, p, q]$$

This means that according to the theorem, the prenex form q might depend on the type α! This problem no longer arises if instead we prove:

$$\forall p.\, \forall M : (\alpha)interpretation.\, E[M, p, Prenex(p)]$$

Prenexing and Skolemization can't quite be defined as primitive recursions over the structure of formulas. For example it can happen that we need to rename variables, so we define a function on a formula in terms of *a substitution instance of* one of its parts. It's easier, then, to argue by wellfounded induction on the size of a formula (as do K&K), which we define as:

```
|- (size False = 1) ∧
   (size (Atom p l) = 1) ∧
   (size (q --> r) = size q + size r) ∧
   (size (!!x q) = 1 + size q)
```

We can define the recursive functions we want by using this, together with the fact that the formulas `!!x p` and `??y q` are always distinct. (This needs to be proved since the latter is not a primitive notion – for example it isn't true that `Not p` and `??y q` are always distinct.) We use the fact that substitution doesn't change the size of a formula, an easy structural induction:

```
|- size (formsubst i p) = size p
```

To prenex an arbitrary formula, we need to be able to prenex falsity, atoms, implications and universal quantifications, for then the prenexability of an arbitrary formula follows by structural induction. Falsity and atoms are trivial; they are already prenex. Universal quantifiers are also easy: just prenex the body. This leaves implication as the only nontrivial case, with the overall definition of the prenexing procedure being:

```
|- (Prenex False = False) ∧
   (Prenex (Atom a l) = Atom a l) ∧
   (Prenex (p --> q) = Prenex_left (Prenex p) (Prenex q)) ∧
   (Prenex (!!x p) = !!x (Prenex p))
```

This requires the function `Prenex_left`, which is supposed to prenex an implication assuming that its antecedent and consequent are already prenex. This is its turn is defined by the following recursion equations, shown to be admissible as indicated above:

```
|- (∀p x q. Prenex_left (!!x q) p =
            let y = VARIANT(FV(!!x q) UNION FV(p)) in
            ??y (Prenex_left (formsubst (valmod (x,V y) V) q) p)) ∧
   (∀p x q. Prenex_left (??x q) p =
            let y = VARIANT(FV(??x q) UNION FV(p)) in
            !!y (Prenex_left (formsubst (valmod (x,V y) V) q) p)) ∧
   (∀p q. qfree q ⇒ (Prenex_left q p = Prenex_right q p))
```

Note that the intimidating-looking `formsubst (valmod (x,V y) V)` simply substitutes variable `y` for variable `x`, leaving other variables unchanged. (Perhaps it would be worth defining a shorter form for this special case.) Apart from performing variable renaming, this simply pulls the quantifiers in the consequent outwards one by one. When the consequent is quantifier-free, an analogous function `Prenex_right` is called to do likewise for the antecedent: It is now straightforward to prove successively that the three prenexing functions `Prenex_right`, `Prenex_left` and `Prenex` do the right thing in the appropriate situation, most notably:

```
|- prenex(Prenex p) ∧
   (FV(Prenex p) = FV(p)) ∧
   (language {Prenex p} = language {p}) ∧
   ∀M v. ¬(Dom M = EMPTY) ⇒ (holds M v (Prenex p) = holds M v p)
```

This says that the result of applying `Prenex` to a formula `p` is indeed prenex, has the same language and free variables as `p` and is logically equivalent to it. Note that for the last part we need to assume nonemptiness of the domain. Prenex normal forms do not in general exist without this restriction, since any prenex formula containing quantifiers is either trivially true or false in an empty domain, regardless of the valuation.

Next we come to Skolem normal forms. The idea here is to replace a formula $\exists y.\, P[x_1, \ldots, x_n, y]$ with free variables $x_1, \ldots, x_n$ by $P[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]$ where f is a fresh function symbol, not originally appearing in P. While not logically equivalent in general, the Skolemized form implies the original, while any model of the original can be extended to a model of the Skolemized form by picking the appropriate interpretation of f, or more precisely in our framework, of (f, n). The starting point is a HOL definition of just this procedure:

```
|- Skolem1 f x p =
     formsubst (valmod (x,Fn f (MAP V (list_of_set(FV(??x p))))) V) p
```

Here `list_of_set`, not surprisingly, converts a finite set to a list. It's defined as an inverse of the operation of converting a list to a set:

```
|- (set_of_list [] = {}) ∧
   (set_of_list (CONS h t) = h INSERT (set_of_list t))

|- list_of_set s = εl. (set_of_list l = s) ∧ (LENGTH l = CARD s)
```

We're mainly interested in Skolemizing a formula already in prenex form, so in the main theorem about `Skolem1`, we make this assumption. The derivation of the main theorem is a bit lengthy, with lots of details to get right, but intuitively obvious. The most difficult part is showing how to extend a model of the existing formula to a model of the Skolemized form. We pick an appropriate denotation of the new function symbol, namely:

```
λg zs. if (g = f) ∧ (LENGTH zs = CARD(FV(??x p)))
       then εa. a IN Dom(M) ∧
                holds M
                 (valmod (x,a)
                    (ITLIST valmod
                      (MAP2 (λx a. (x,a)) (list_of_set(FV(??x p))) zs)
                      (λz. εc. c IN Dom(M)))) p
       else Fun(M) g zs
```

where:

```
|- (MAP2 f [] [] = []) ∧
   (MAP2 f (CONS h1 t1) (CONS h2 t2) = CONS (f h1 h2) (MAP2 f t1 t2))

|- (ITLIST f [] b = b) ∧
   (ITLIST f (CONS h t) b = f h (ITLIST f t b))
```

The definition looks complicated, but corresponds to intuition. We are defining the interpretation of function `g` on argument (list) `zs`. If it isn't the new function, `f` with the right arity, then we just take the denotation given by `M`, i.e. we don't change the interpretation of other functions. Otherwise, we have that there exists some a satisfying (the interpretation of) `p`; we make the above definition to correspond to unfolding the definition of validity in this case. The valuation constructed maps each free variable – which is an argument of the Skolem function – to the appropriate object of the model.

However, we want to apply Skolemization repeatedly to get rid of all existential quantifiers, and moreover, for later use we need to be able to apply it to a set of formulas 'in parallel'. K&K (p. 23) say 'we assume that the function letters added to $\mathcal{L}(\mathcal{E})$ for different formulas F are different'. However it takes quite a bit of work to make sure of this. Roughly speaking, starting from a set S of formulas, we just need to take an additional set of distinct function symbols with size equal to the number of existential quantifiers in S. In something like ZF set theory this presents no serious difficulties. It's more awkward in HOL since we can only add function symbols from the same underlying type that was used originally, at present `:num`. It's perfectly possible that the original set of formulas included every possible function symbol, leaving us no more to add. So we can't literally start with exactly the same set of function symbols. First we need to map them into a type with room for additions. In fact we use the same type `:num` starting with the following injective pairing function (we'll write $< x, y >$ informally instead of `NUMPAIR x y`)

```
|- NUMPAIR x y = (2 EXP x) * (2 * y + 1)
```

and the corresponding destructors

```
|- NUMFST(NUMPAIR x y) = x

|- NUMSND(NUMPAIR x y) = y
```

A similar approach works for any infinite type since one can define an injective pairing. Now we take a formula p and shift up all the function symbols from (k, n) to $(< 0, k >, n)$.

```
|- (bumpterm (V x) = V x) ∧
   (bumpterm (Fn k l) = Fn (NUMPAIR 0 k) (MAP bumpterm l))

|- (bumpform False = False) ∧
   (bumpform (Atom p l) = Atom p (MAP bumpterm l)) ∧
   (bumpform (q --> r) = bumpform q --> bumpform r) ∧
   (bumpform (!!x r) = !!x (bumpform r))
```

Obviously we can make a corresponding change in the interpretation:

```
|- bumpmod(M) = Dom(M),(λk zs. Fun(M) (NUMSND k) zs),Pred(M)
```

so that

```
|- holds M v (bumpform p) = holds (unbumpmod M) v p
```

Now we have all function symbols of the form $(< m + 1, k >, n)$ to use as Skolem functions. Using `NUMPAIR`, we define a 'Gödel numbering' of formulas, mapping a formula p to a number $\ulcorner p \urcorner$, written in HOL as `num_of_form`. Now we use $(< 1 + \ulcorner p \urcorner, k >, n)$ for the k^{th} Skolem function (with arity n) used in Skolemizing a formula p. We will not show the HOL statement of the theorem (20 lines), but it says that the Skolemized version of a formula is universal, with the same free variables and a language only changed by adding the appropriate Skolem functions, that any model of a formula extends to a model of its Skolemized form, and that the Skolemized form logically implies the original. From this we distil the key fact needed later: a set of formulas is satisfiable iff the Skolemized form is. We add a final pass to strip off all the universal variables:

```
|- (specialize False = False) ∧
   (specialize (Atom p l) = Atom p l) ∧
   (specialize (q --> r) = q --> r) ∧
   (specialize (!!x r) = specialize r)

|- SKOLEM p = specialize(SKOLEMIZE p)
```

and the theorem looks like this:

```
|- (∃M. ¬(Dom M :A->bool = EMPTY) ∧
        interpretation (language s) M ∧ M satisfies s) =
   (∃M. ¬(Dom M :A->bool = EMPTY) ∧
        interpretation (language {SKOLEM p | p IN s}) M ∧
        M satisfies {SKOLEM p | p IN s})
```

4.3 Canonical Models

Thanks to Skolemization, we can for many purposes restrict ourselves to considering quantifier-free formulas. In this case, they can also be treated as formulas of propositional logic, and the relation between these two views is central to what follows. First of all, given an interpretation `M` and valuation `v` we can choose a corresponding propositional valuation:

```
|- prop_of_model M v (Atom p l) = holds M v (Atom p l)
```

'corresponding' in the following precise sense:

```
|- qfree(p) ⇒ (pholds (prop_of_model M v) p = holds M v p)
```

That is, a quantifier-free formula holds in a given `M` and `v` iff it holds as a formula of propositional logic under valuation `prop_of_model M v`. Conversely, given a propositional valuation, we can find a corresponding first order interpretation and valuation. We are at liberty to interpret the atomic formulas to match the required propositional valuation, provided that the interpretation of function symbols is such that the interpretation cannot map distinct terms to identical objects of the domain. There are plenty of ways of avoiding this, the simplest being to take as the domain (a suitable subset of) the set of terms, and interpret function symbols 'as themselves', i.e.

```
|- canon_of_prop L = terms(FST L),Fn,λp l. d(Atom p l)
```

There is some freedom over how to choose the domain of the model. We take the set of all terms restricted to a certain first order language `L`, which is defined inductively by:

```
|- (∀x. terms fns (V x)) ∧
   (∀f l. (f,LENGTH l) IN fns ∧ FORALL (terms fns) l
          ⇒ terms fns (Fn f l))
```

In general we say that an interpretation is a *canonical* interpretation of a language if the domain and interpretations of function symbols are defined as in the above particular case:

```
|- canonical L M = (Dom M = terms (FST L)) ∧ (∀f. Fun(M) f = Fn f)
```

Under the identity valuation `V`, terms are literally interpreted as themselves, and so `canon_of_prop` does indeed work as hoped, i.e.

```
|- qfree p ⇒ (holds (canon_of_prop L d) V p = pholds d p)
```

In particular, the theorem about `prop_of_model` tells us that a propositional tautology is first order valid in all interpretations, while this one tells us that if a formula holds in all (or even just all canonical) interpretations, then it is a propositional tautology.

```
|- qfree(p) ∧ (∀d. pholds d p) ⇒ ∀M v. holds M v p

|- qfree(p) ∧ (∀C v. canonical(language {p}) C ⇒ holds C v p)
   ⇒ ∀d. pholds d p
```

This extends to an arbitrary set of formulas. However none of this works if we consider not validity, but satisfiability. If a formula is first order satisfiable it is propositionally satisfiable:

```
|- (∀p. p IN s ⇒ qfree p) ∧ M satisfies s ∧ valuation(M) v
   ⇒ (prop_of_model M v) psatisfies s
```

But if it's propositionally satisfiable, we only know that it is satisfiable *under the identity valuation*, not more generally. For example, the formula $P(x) \wedge \neg P(y)$ is propositionally satisfiable but not first order satisfiable. However, it's easy to prove a slight variant of the second theorem:

```
|- qfree p
   ⇒ (holds (canon_of_prop L d) v p = pholds d (formsubst v p))
```

This tells us that a formula is first order satisfiable iff all its substitution instances (in the appropriate language) are propositionally satisfiable:

```
|- (∀p. p IN s ⇒ qfree p) ∧
   d psatisfies {formsubst v p | p IN s ∧ ∀x. v x IN terms(FST L)}
   ⇒ (canon_of_prop L d) satisfies s
```

Moreover, note that we actually have something a bit stronger: if all substitution instances are propositionally satisfiable, then it has a *canonical* first order model. This fact, not isolated by K&K but hinted at in their proof of the Compactness theorem, is the key to getting the main metatheorems. First let us note that an interpretation satisfies all substitution instances of a set of formulas w.r.t. an arbitrary language iff it satisfies the set itself.

```
|- interpretation(language t) M ∧
   ⇒ (M satisfies
        {formsubst i p | p IN s ∧ ∀x. i x IN terms(FST(language t))} =
      M satisfies s)
```

Now we get a simple proof of the Compactness and Löwenheim-Skolem theorems together, whereas K&K establish them by separate arguments. If all finite subsets of a set of quantifier-free formulas have a model then the set as a whole has a (canonical) model:

```
|- (∀p. p IN s ⇒ qfree p) ∧
   (∀t. FINITE t ∧ t SUBSET s
        ⇒ ∃M. interpretation(language ss) M ∧
                ¬(Dom(M) = EMPTY) ∧ M satisfies t)
   ⇒ ∃C. interpretation (language ss) C ∧
           canonical (language ss) C ∧ C satisfies s
```

Once again we have a certain liberty over the language `ss`. Note that all the models of finite subsets are based on a particular type – HOL forces us to make such a restriction. However, precisely because we have as a corollary that every set of formulas that has a model has a canonical model, this doesn't amount to a genuine weakening of the theorem as normally stated. Now we can use Skolemization to lift this theorem to the class of all formulas, not just quantifier-free ones. We reason as follows. If all finite subsets T of S are satisfiable, then so are the Skolemized forms T^*, and so all finite subsets of S^*, which must be contained in such a set. By the above theorem, S^* has a canonical model, and this is also a model of S. This last step isn't quite true in our formalization; it's really a model of the modified form of S with the function symbols shifted. So when we reverse the transition, the model isn't actually canonical. We could have modified the construction of the canonical model appropriately to make this work, but it doesn't seem worth it: the interesting thing about the model is really that it's countable.

We separate out the fact that every set of formulas that has a model has a model with domain a subset of the set of terms, simply using the fact that if a set is satisfiable, so are all finite subsets of it. By applying the Gödel numbering on terms used earlier, we can map the model into the natural numbers and obtain the usual form of the Löwenheim-Skolem theorem.

```
|- interpretation (language s) M ∧
   ¬(Dom M :A->bool = EMPTY) ∧
   M satisfies s
   ⇒ ∃N. interpretation (language s) N ∧
           ¬(Dom N :num->bool = EMPTY) ∧ N satisfies s
```

Finally, using the same lemmas about substitution instances, we easily obtain the Uniformity (aka Skolem-Gödel-Herbrand) theorem, i.e. that if an purely existential term is provable, some disjunction of substitution instances of the body is also provable, or equivalently, is a propositional tautology. The reasoning is straightforward; we'll sketch it with one variable for clarity. If $\exists x.\ p$ is valid then $\forall x.\ \neg p$ is unsatisfiable. Hence the set of all substitution instances $\neg p[t/x]$ is propositionally unsatisfiable; by propositional compactness, so is some finite

subset, and hence a finite conjunction $\neg p[t_1/x] \wedge \cdots \wedge \neg p[t_n/x]$. But this means that the negation $p[t_1/x] \vee \cdots \vee p[t_n/x]$ is a propositional tautology and first order valid. In HOL we prove the full theorem in this form:

```
|- qfree p ∧ (∀C v. ¬(Dom C = EMPTY) ∧ valuation C v
                    ⇒ holds C v (ITLIST ?? xs p))
   ⇒ ∃is. (∀i x. MEM i is ⇒ terms (FST (language {p})) (i x)) ∧
          (∀d. pholds d
               (ITLIST (||) (MAP (λi. formsubst i p) is) False))
```

The iterated uses of quantifiers and disjunctions are expressed using the standard list operation `ITLIST` defined earlier.

5 Related Work

Most related work in theorem provers is more heavily syntactic, dealing with proof theory. However, Persson [8] formalizes in the ALF prover a constructive completeness proof for intuitionistic logic w.r.t. models based on formal topology.

6 Conclusions

Our original goal has been achieved: we have machine-checked the proofs of the theorems given here, and have significantly improved the original proofs. While applications weren't at the front of our mind, we have since used the above work as the basis of a construction of the hyperreals. First we extended the Compactness theorem to first logic with equality (once more following K&K). This means restricting ourselves to *normal* models where binary predicate 0 is interpreted as equality. Then we used this to show that there is a model of the set of all first order statements true in the reals which also has infinite elements. This could be used to give a foundation for theorem proving in nonstandard analysis, though we haven't pursued that.

Our paper can be read as strong support for adding first-class type quantifiers to HOL, as proposed by Melham [7]. More generally, though, types make little positive contributions in this area, bearing out the commonplace feeling that types tend to become a hindrance in more abstract parts of mathematics.

Just whether the awkwardness of choosing Skolem functions indicates a failure in HOL (logic or system) or a genuine gap in K&K is a matter of opinion. Certainly, it's quite intuitive that one can pick the Skolem functions independently, but it's interesting that the reasoning we did almost duplicates the kind of constructions in Henkin-style completeness proofs [2]. These tend to look ugly because one needs to add constants to the language (to act as 'witnesses' for existential assertions), but this yields new existential formulas so the procedure needs to be iterated. The K&K proofs avoid this nicely by Skolemizing once and for all. Yet when doing this, we still need to pay attention to the same kinds of issues.

References

1. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, **34**, 381–392, 1972.
2. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
3. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison (eds.), *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, Volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, pp. 173–190. Springer-Verlag, 1996.
4. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
5. P. T. Johnstone. *Notes on Logic and Set Theory*. Cambridge University Press, 1987.
6. G. Kreisel and J.-L. Krivine. *Elements of mathematical logic: model theory* (Revised second ed.). Studies in Logic and the Foundations of Mathematics. North-Holland, 1971. First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles' published by Dunod, Paris in 1964.
7. T. F. Melham. The HOL logic extended with quantification over type variables. In L. J. M. Claesen. and M. J. C. Gordon. (eds.), *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Volume A-20 of *IFIP Transactions A: Computer Science and Technology*, IMEC, Leuven, Belgium, pp. 3–18. North-Holland, 1992.
8. H. Persson. *Constructive Completeness of Intuitionistic Predicate Logic: A Formalisation in Type Theory*. Licentiate thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, Sweden, 1996.
9. R. Pollack. *The theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. Ph. D. thesis, University of Edinburgh, 1994.
10. D. Prawitz. *Natural deduction; a proof-theoretical study*, Volume 3 of *Stockholm Studies in Philosophy*. Almqvist and Wiksells, 1965.
11. P. Rudnicki. An overview of the MIZAR project, 1992. Available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.
12. S. Shapiro. *Foundations without Foundationalism: A case for second-order logic*. Number 17 in Oxford Logic Guides. Clarendon Press, 1991.
13. A. Stoughton. Substitution revisited. *Theoretical Computer Science*, **17**, 317–325, 1988.
14. A. Trybulec. The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, **6**, 136–140, 1978.

This article was typeset using the LaTeX macro package with the LLNCS2E class.