Specialization Seminar

First-order Logic (2)
Mechanizing Herbrand's Theorem

Peer David
david.peer@student.uibk.ac.at

12 February 2017

Supervisor: Julian Nagele

Abstract

This report is a summary of the Chapters 3.8 to 3.16 of the book "*Handbook of Practical Logic and Automated Reasoning*". The Chapters of the book describe an automatic way to prove validity of first-order formulas.

Contents

1 Introduction

Automatic proving system are used in different domains. One example where such systems are used is the logical programming language Prolog. In this report we will show, how we can implement automatic proving systems and use these systems to execute Prolog programs. To accomplish this task, we reduce first-order satisfiability to propositional satisfiability using the Herbrand model. Afterwards we introduce an automatic way to prove validity of first-order formulas from 1960, called the Gilmore Procedure. In the next sections we will see two different methods (tableaux and resolution) that improve the original Gilmore Procedure. Different improvements of resolution will be provided in section 7. At the end of the report we introduce the logical programming language Prolog where we use such an automatic proving system.

Note: All definitions, theorems and corollaries are from [2]. The original name of [2] is added in parentheses. If we used a definition of another reference, the original source is cited explicitly.

2 Preliminaries

OCaml elements such as list names or functions used in definitions, lemmas or corollaries are highlighted in `this` style. The letter σ represents a substitution and `tsubst` σ t is the result of applying the substitution σ to the term t. We denote negation, conjunction, disjunction and implication by $\neg$, $\wedge$, $\vee$ and $\rightarrow$. We write $P[x_1, ..., x_n]$ to indicate that the formula P contains the variables $x_1, ..., x_n$, and $P[t_1, ..., t_n]$ to denote the result obtained by replacing $x_1, ..., x_n$ by $t_1, ..., t_n$. Names of procedures or rules which are introduced are written in *italic*. Notes are included in cursive style and start with *Note:* Implementations that we used in this report are provided by [2] and can be downloaded from `http://www.cl.cam.ac.uk/~jrh13/atp/OCaml.tar.gz`. *Note: The implementation of* ***`simple_resolution`*** *is not included in the tar file and is included in Appendix A.*

Before we can start with an automatic way to prove validity of first-order formulas, we reduce first-order satisfiability to propositional satisfiability. To accomplish this task, we introduce the *Herbrand model*.

Definition 2.1 (Herbrand Domain)**.** The Herbrand domain for a particular first-order language is the set of all ground terms of that language.

Definition 2.2 (Herbrand Base [4])**.** The Herbrand base is the set of all ground atoms formed using elements of the Herbrand domain as arguments.

Definition 2.3 (Herbrand Interpretation [4])**.** A Herbrand interpretation is a interpretation whose

- domain is the the Herbrand domain,
- functions are the identity function,

- predicates are a subset of the Herbrand base

Definition 2.4 (Herbrand Model [4])**.** A Herbrand model of a set of formulas is a Herbrand interpretation that is a model of every formula in the set.

If we build a Herbrand domain for a specific language, we create all ground terms using functions and constants of this language. If there exists no constant, we add a constant c.

Now we can introduce two important theorems:

Theorem 2.5 (3.23)**.** *A quantifier-free formula p is first-order satisfiable iff the set of all its ground instances is (propositionally) satisfiable.*

Theorem 2.6 (3.24)**.** *A quantifier-free formula has a model (i.e. is satisfiable) iff it has a Herbrand model.*

Theorem 2.5 shows, that we can use a Herbrand model (the Herbrand domain is the set of all ground instances) to prove first-order satisfiability of formulas. Theorem 2.6 states, that if a quantifier-free formula has a model, there always exists a Herbrand model. So we can create an automatic way to prove satisfiability of first-order formulas using the Herbrand model.

3 Mechanizing Herbrand's Theorem

To check if a first-order formula is satisfiable, we test propositional satisfiability of the set of all ground atoms (Theorem 2.5). Usually, there are infinitely many ground instances. So we use the compactness theorem to create larger and larger sets of ground instances and test these sets for propositional satisfiability. *Note: If the formula is satisfiable, this process may never terminate.*

Theorem 3.1 (Compactness [3])**.** *A set Γ of sentences of predicate logic is satisfiable iff all finite subsets of Γ are satisfiable.*

If we combine Theorem 2.5 and Theorem 3.1, we can conclude:

Theorem 3.2. *A quantifier-free formula is first-order satisfiable iff all finite sets of ground instances are (propositionally) satisfiable.*

The inverse of Theorem 3.2 is:

Corollary 3.3. *A quantifier-free formula p is first-order unsatisfiable iff some finite set of ground instances is (propositionally) unsatisfiable.*

With Corollary 3.3 we can create a procedure, to check, if a formula is valid. For example we can check, if the formula $p = \forall x.\exists y.P(x) \rightarrow P(y)$ is valid:

1. Skolemize $\neg p$ (to check validity): $\forall x.P(x) \wedge \neg P(f_y(x))$
2. Build the Herbrand Domain: $c, f_y(c), f_y(f_y(c)), ...$

3. The conjunction of all instances must be propositionally satisfiable
   a) $P(c) \wedge \neg P(f_y(c))$
   b) $P(c) \wedge \boldsymbol{\neg P(f_y(c)) \wedge P(f_y(c))} \wedge \neg P(f_y(f_y(c))$
   c) **Unsatisfiable**
4. The Forumla is valid

In step 3.a) we used the first finite set $\{c\}$ of ground instances. In this step we can not conclude, that the formula is unsatisfiable. So in step 3.b) we created a larger set $\{c, f_y(c)\}$ of ground instances. The conjunction of all instances must be propositionally satisfiable, but the instance in b) is unsatisfiable because it contains the complementary literals $\neg P(f_y(c))$ and $P(f_y(c))$. Using the compactness theorem we can conclude, that the formula $\neg p$ is unsatisfiable and p is valid.

One implementation of this procedure is the *Gilmore Procedure*. In this procedure for a formula in DNF, we enumerate larger and larger sets of ground instances and check at each stage every disjunct of the DNF for complementary literals. Another implementation is the *Davis-Putnam Procedure*. Formulas are in CNF rather than DNF and each time we create new instances we check for unsatisfiability using DPLL. The algorithms are provided by [2] and can be executed as follows:

Listing 1: Execution of the Gilmore and Davis-Putnam Procedure.

```
let testfm = <<exists x. exists y. forall z.
  (F(x,y) ==> (F(y,z) /\ F(z,z))) /\
  ((F(x,y) /\ G(x,y)) ==> (G(x,z) /\ G(z,z)))>>;;

# gilmore testfm;;
...
11 ground instances tried; 569856 items in list
Stack overflow during evaluation (looping recursion?).

# davisputnam testfm;;
...
410 ground instances tried; 1230 items in list
- : int = 411
```

In this example we can see, that it was not possible to check validity with the Gilmore Procedure for our testfm. The problem is, that we create a DNF which uses lots of memory. So the call ends in a stack overflow. The Davis-Putnam Procedure needs 411 steps to finish, getting slower and slower at each step by creating bigger and bigger sets of ground instances.

4 Unification

The Davis-Putnam Procedure avoids the memory explosion of the Gilmore Procedure, but we still create larger and larger sets of ground instances and so the procedure becomes slower at each step. Instead of blindly trying all possibilities, we can also work with

uninstantiated formulas and create the instances only when we need them. For example if we try to find complementary literals in the two clauses

$$P(x, f(y)) \vee Q(x, y)$$
$$\neg P(g(u), v)$$

we set $x = g(u)$ and $v = f(y)$ to get the instantiation

$$P(g(u), f(y)) \vee Q(g(u), y)$$
$$\neg P(g(u), f(y))$$

If we apply the resolution rule (see Section 6) we get the new clause $P(g(u), f(y))$. To match two different terms with such an instantiation, we need a procedure called *unification*.

Note: In the following definitions we use the Ocaml implementation `tsubst sfn tm` *to substitute terms for variables (*`tm`*) in another term or formula* `sfn`*.*

Definition 4.1 (3.27)**.** Given a set of pairs of terms $S = \{(s_1, t_1), ..., (s_n, t_n)\}$ a unifier of the set S is an instantiation σ such that `tsubst` σ $s_i =$ `tsubst` σ $t_i \quad forall\ i = 1, ..., n$.

Definition 4.2 (more general)**.** We say that an instantiation σ is more general than another one τ, and write $\sigma \leq \tau$, if there is some instantiation δ such that `tsubst` $\tau =$ `tsubst` $\delta \circ$ `tsubst` σ.

Definition 4.3 (MGU)**.** We call σ a most general unifier (MGU) of S if

1. σ is a unifier of S,
2. for every other unifier τ of S, we have $\sigma \leq \tau$.

MGUs are not necessarily unique but they differ only up to a permutation of vaiable names. For instance $\{(x, y)\}$ has two different MGUs namely $x \to y$ and $y \to x$.

An implementation of the unification algorithm is provided by [2]. The algorithm has the following properties:

1. If there exists no instantiation to unify two formulas, the algorithm fails.
2. If there exists an instantiation, the algorithm returns the MGU.
3. The algorithm terminates in the case of failure and success.

Here we can see an example, how we can use the implementation of the unification algorithm:

Listing 2: Execution of unify_and_apply

```
# let fm1 = <<|f(x, g(y), x) |>>;;
# let fm2 = <<|f(z,g(u),h(u))|>>;;
# unify_and_apply [fm1, fm2];;

- : (term * term) list =
[(<<|f(h(u),g(u),h(u))|>>, <<|f(h(u),g(u),h(u))|>>)]
```

In the next section we will see a method called tableaux, which uses unification to create instances only, if we need them.

5 Tableaux

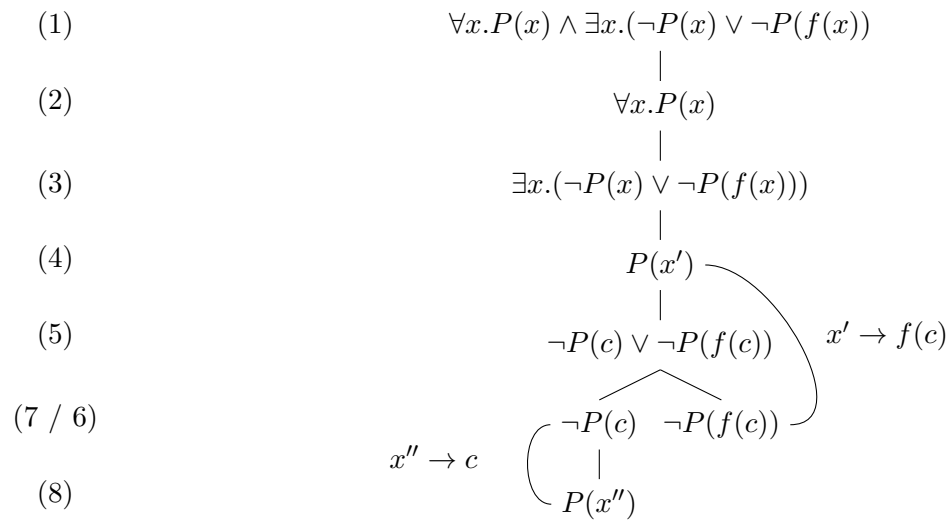
If a formula $P[x_1, ..., x_n]$ is unsatisfiable, there are finitely many ground instances such that $P[t_1^1, ..., t_n^1] \wedge ... \wedge P[t_1^k, ..., t_n^k]$ is unsatisfiable. Instead of creating those ground instances, we replace the variables $x_1, ..., x_n$ with tuples of distinct variables: $P[z_1^1, ..., z_n^1] \wedge ... \wedge P[z_1^k, ..., z_n^k]$. So we know that there exists an instantiation θ, that maps z_i^j to t_i^j. But this means, that also the uninstantiated DNF must contain two unifiable complementary literals.

In this way, we never have to generate the ground terms, but rather let the necessary instantiations emerge gradually by need. In *Analytic Tableaux* we create a tree of subformulas, trying to lead to a contradiction (by unifying complementary literals) at every branch. If every branch reaches a contradiction, we know that the whole formula is unsatisfiable. With the following rules from [1] and [2], we can create such a proof tree:

1. $p \wedge q$ - separately assume p and q
2. $p \vee q$ - perform two refutations, one assuming p and one assuming q
3. $\forall x.P[x]$ - introduce a new variable x' and assume $P[x']$, but also keep the original $\forall x.P[x]$ in case multiple instances are needed
4. $\exists x.P[x]$ - introduce $P[f(x_1, ..., x_n)]$ where f is a new function symbol and $x_1, ..., x_n$ the free variables of P

In the following example we will see, how we can use the rules of analytic tableaux to check unsatisfiability of the formula $\forall x.P(x) \wedge \exists x.(\neg P(x) \vee \neg P(f(x))$:

Example 5.1. Tableaux example from [1]:



In (1) we can see the original formula. We use rule 1 and separately assume $\forall x.P(x)$ and $\exists x.(\neg P(x) \vee \neg P(f(x)))$. In (4) we use the forall-rule and introduce a new variable x'. In (5) we use the exists-rule and introduce a new function with the arity of the number of free variables. The number of free variables is 0, so we introduce a new constant c. In (7) and (6) we performed two different refutations, one for $\neg P(c)$ and one for $\neg P(f(c))$. The branch of (6) can be closed, because there exists a substitution, which unifies (4) and (6) leading to a contradiction. We already unified $P(x')$, so we cannot use it again in (7). In (8) we use the forall-rule again and introduce a new variable x'' to create a contradiction and so we can also close the second branch.

In listing Listing 3 we can see the implementation provided by [2]:

Listing 3: Analytic Tableaux implementation in OCaml.

```
let rec tableau (fms,lits,n) cont (env,k) =
    if n < 0 then failwith "no␣proof␣at␣this␣level" else
    match fms with
    [] -> failwith "tableau:␣no␣proof"
    | And(p,q)::unexp ->
        tableau (p::q::unexp,lits,n) cont (env,k)
    | Or(p,q)::unexp ->
        tableau (p::unexp,lits,n) (tableau (q::unexp,lits,n) cont) (env,k)
    | Forall(x,p)::unexp ->
        let y = Var("_" ^ string_of_int k) in
        let p' = subst (x |=> y) p in
        tableau (p'::unexp@[Forall(x,p)],lits,n-1) cont (env,k+1)
    | fm::unexp ->
        try tryfind (fun l -> cont(unify_complements env (fm,l),k)) lits
        with Failure _ -> tableau (unexp,fm::lits,n) cont (env,k);;
```

If we look at the algorithm provided by [2] we can see, that the exists-rule is not implemented. The reason is, that we want to proof validity of formulas and skolemize the negated formula, before we apply Analytic Tablaux. So there are no existential quantifiers in the formula and so we don't need an implementation for this rule.

If we execute the Tableaux implementation we can see, that tableaux needs 7 step to show validity of our test formula:

Listing 4: Execution of tableaux.

```
# tab testfm;;
Searching with depth limit 0
...
Searching with depth limit 6
Searching with depth limit 7
- : int = 7
```

6 Resolution

In the previous chapter we have seen, that the variable instantiations in the analytic tableaux procedure need to be propagated throughout the proof. Such a type of procedure is called a *global method*. In this section we will see a *local method* called resolution, where variable instantiations are not propagated throughout the proof.

Suppose that two clauses $C[x_1, ..., x_n]$ and $D[y_1, ..., y_n]$ have instances to which propositional resolution is applicable, say:

$$C[x_1, ..., x_n] = ... \vee P(s_1, ..., s_m) \vee ...$$

and

$$D[y_1, ..., y_n] = ... \vee \neg P(s'_1, ..., s'_m) \vee ...$$

such that when the appropriate ground instantiation θ is applied, it unifies the set $S = \{(s_i, s'_i) \mid i = 1, ..., m\}$. This allows us to apply the *resolution rule*: from two clauses $p \vee C_1$ and $\neg p \vee C_2$ we deduce the conclusion $C_1 \vee C_2$. To unify both sets, we use the MGU instead of θ. But it is not always possible to derive a result using the MGU. If we apply the resolution rule for example to $\{\{\neg P(x, x), \neg P(c_b, x)\}, \{P(x, x), P(c_b, x)\}\}$, we only create new tautologies, which are not useful for our automatic proving system. So we unify some subset of the literals in the same clause (called factoring):

$$\{\neg P(x, x), \neg P(c_b, x)\} \to \{\neg P(c_b, c_b)\}$$
$$\{P(x, x), P(c_b, x)\} \to \{P(c_b, c_b)\}$$

Theorem 6.1 (3.29)**.** *If a set S of first-order clauses is unsatisfiable, the empty clause is derivable using resolution.*

Now we can create an algorithm which is refutation complete (if the set S is unsatisfiable, we derive the empty clause):

1. Split a set S of clauses into list of `used` and `unused`
2. Move the top clause from `unused` to `used`
3. Generate all possible resolvents of the selected clause with all clauses from `used`
4. Append all new generated clauses to `unused`
5. Finish when `unused` is empty (satisfiable - *proof failed*) or the empty clause is derived (unsatisfiable)

If we execute this algorithm we can see, that resolution needs 84 step to show validity of the formula `testfm` used in section 3. *Note: The code for simple_resolution is not provided by the book. It is added in Appendix A.*

Listing 5: Execution of resolution without subsumption and replacement

```
# simple_resolution testfm;;
...
83 used; 483 unused.
84 used; 488 unused.
- : bool list = [true]
```

In the next Section we will see, how we can improve this algorithm using subsumption and replacement.

7 Subsumption and Replacement

Some problems are very difficult for the resolution procedure to solve and result in the generation of thousands of clauses without leading to a solution. So we will introduce *subsumption* which allows us, to exclude newly generated clauses (or to remove already existing clauses), which are unnecessary for the proof.

Definition 7.1 (Subsumption)**.** A first-order clause C subsumes another clause D, written $C \leq_{ss} D$, if there is some instantiation θ such that `tsubst` θ C is a subset of D.

Now we can use subsumption to select always those clauses, which are not subsumed by any of its ancestors:

Theorem 7.2 (3.33)**.** *If C is derivable by resolution from hypotheses S, then there is a resolution derivation of some C' with $C' \leq_{ss} C$ from S in which no clause is subsumed by any of its ancestors.*

Using Theorem 7.2 we can improve the implementation used in Listing 5 in three different ways:

1. *Forward deletion*: If a newly generated clause is subsumed by one already present, discard the newly generated clause;

2. *Backward deletion*: If a newly generated clause subsumes one already present, discard the one already present;

3. *Backward replacement*: If a newly generated clause subsumes one already present, replace the one already present by the newly generated one.

If we want to use these improvements, we have to take care about the implementation of the already existing resolution algorithm 6:

We can use *forward deletion* to discard newly generated clauses only, if the subsuming clause is in the `unused` list. If we delete a newly generated clause also, when the subsuming clause is in `used`, there is no new clause which will be resolved with clauses in `used`. For that reason the new resolution algorithm will implement only forward deletion with clauses, that are contained in `unused`.

If we use *backward deletion* it is possible, that the proving algorithm is getting slower: If a newly generated clause subsumes one clause in the `unused` list, this clause will be deleted and the new clause will be appended to the list. This means that the clause needs longer to get to the top of the list. If we look at Listing 6 we can see, that always the top clause is used to generate all possible resolvents. So it takes longer for the clause to be used and so the algorithm becomes slower. For that reason, the improved resolution algorithm will not contain backward deletion.

There is no problem using *backward replacement*. So backward replacement is implemented for both lists in the algorithm:

Listing 6: Implementation of resolution with subsumption and replacement.

```
let incorporate gcl cl unused =
    if trivial cl or exists (fun c -> subsumes_clause c cl) (gcl::unused)
    then unused
    else replace cl unused;;
```

If we look at the algorithm provided by the book, we can see that

1. We remove all tautologies: `if trivial cl [...] then unused [...]`
2. We only delete newly generated clauses, if the subsuming clause is in the `unused` list: `if [...] or exists (fun c -> subsumes_clause c cl) (gcl::unused) then unused [...]`
3. In all other cases we replace the new clause: `[...] else replace cl unused`

If we execute the resolution with subsumption and replacement algorithm we can see, that we need 7 steps instead of 84:

Listing 7: Execution of resolution with subsumption and replacement.

```
# resolution testfm;;
...
6 used; 3 unused.
7 used; 2 unused.
val davis_putnam_example : bool list = [true]
```

In the next chapter we will see an example, where we can use such automatic proving systems.

8 Horn Clauses and Prolog

We know, that SAT solving is an NP-complete problem. If we restrict our formulas to Horn formulas, we can create an efficient decision procedure for satisfiability of Horn formulas [3]. In this chapter we will introduce *Horn formulas* and extend the efficient algorithm for propositional logic provided by [3] to first-order logic. Then we will use this algorithm to implement a logical programming language called *Prolog*.

Definition 8.1 (Horn clause)**.** A Horn clause is a clause containing at most one positive literal.

Definition 8.2 (Definite clause)**.** A definite clause is a clause containing exactly one positive literal :

$$\neg P_1 \vee ... \vee \neg P_n \vee Q \text{ with } n \geq 0 \quad \equiv \quad P_1 \wedge ... \wedge P_n \rightarrow Q$$

Definition 8.3 (Goal clause [4])**.** A goal is a clause without a positive literal :

$$\neg P_1 \vee ... \vee \neg P_n \quad \equiv \quad P_1 \wedge ... \wedge P_n \rightarrow \bot$$

Definition 8.4 (Horn formula [3])**.** A Horn formula is conjunction of Horn clauses.

The algorithm provided by [3] is an efficient procedure, to check satisfiability of propositional Horn formulas:

1. Mark $\top$
2. While there is a Horn clause $P_1 \wedge ... \wedge P_n \rightarrow Q$ in the formula such that all $P_1, ..., P_n$ are marked and Q is unmarked, mark Q
3. If $\bot$ is marked, return unsatisfiable else return satisfiable

In the following example we will see, how we can check the satisfiability of a formula. The algorithm also provides the valuation of all atoms to satisfy the formula (witness).

Example 8.5 (Propositional Logic [3])**.**

$$(\boldsymbol{p} \wedge \boldsymbol{q} \wedge w \rightarrow \bot) \wedge (t \rightarrow \bot) \wedge \boldsymbol{r} \rightarrow \boldsymbol{p}$$
$$\wedge \top \rightarrow \boldsymbol{r}) \wedge \top \rightarrow \boldsymbol{q}) \wedge \top \rightarrow \boldsymbol{u}) \wedge \boldsymbol{u} \rightarrow \boldsymbol{s})$$

$$\boldsymbol{p} \quad \boldsymbol{q} \quad \boldsymbol{r} \quad \boldsymbol{s} \quad t \quad \boldsymbol{u} \quad w \quad \bot \quad \top$$

satisfiable: $v(p) = v(q) = v(r) = v(s) = v(r) = T \quad v(t) = v(w) = F$

Now we will introduce an algorithm for first-order logic. Before we can start with the algorithm, we will introduce the *least Herbrand model*, which is used to build a first-order model:

Definition 8.6 (Least Herbrand Model)**.** We construct a Herbrand interpretation M interpreting each n-ary predicate P by $P_M(t_1, ..., t_n) = true$ iff $P_H(t_1, ..., t_n) = true$ for all Herbrand models H of S.

If M is a model of S, we call it the least Herbrand model of S.

But is there always a least Herbrand model for every formula? The following formula is a counterexample: $S = \{P(0) \vee Q(0)\}$

- H_1: $\{P(0)\}$
- H_2: $\{Q(0)\}$
- H_3: $\{P(0), Q(0)\}$

Neither P(0) nor Q(0) holds in all Herbrand models H_n, so M is not a model of S. In this example we can see, that a least Herbrand model does not exist for every formula. But if we restrict our formulas to horn formulas, we can prove the following theorems:

Theorem 8.7 (3.42)**.** *If a set S of Horn clauses is satisfiable, it has a least Herbrand model M which satisifes an atomic formula p iff every Herbrand model of S satisfies p.*

Theorem 8.8 (3.45)**.** *If $P[x_1, ..., x_n]$ is quantifier-free and S is a set of Horn clauses, then $S \models \exists x_1, ..., x_n.P[x_1, ..., x_n]$ iff there is some ground instance such that $S \models P[t_1, ..., t_n]$.*

This gives us a goal-directed way of verifying that some atomic ground formula holds in all models of a set of definite clauses S:

1. Given an initial goal P
2. Search for a clause, such that when instantiated $Q_1 \wedge ... \wedge Q_n \rightarrow P$ has P as conclusion
3. Unify goal P with the head of the clause
4. Show that all subgoals $Q_1, ..., Q_n$ hold in the least model
5. If P holds in the least model there is a specific ground instance that is a consequence of the clauses (Theorem 3.45)

This algorithm is called backchaining and is provided by [2]:

```
let rec backchain rules n k env goals = match goals with
    [] -> env
    | g::gs ->
        if n = 0 then failwith "Too deep" else
        tryfind (fun rule ->
            let (a,c),k' = renamerule k rule in
            backchain rules (n - 1) k'
                (unify_literals env (c,g)) (a @ gs))
        rules;;
```

As we can see we take the first goal and try to unify it with a rule of our program. If no suitable rule exists, the algorithm fails (this is implemented in the function `tryfind`). If we found a rule, we call the backchain recursively appending all our subgoals `a`. If we found an instantiation, which satisfies the initial `goals`, the algorithm returns the ground instances `env`. *Note: Most Prolog implementations provides a backtracking instead of a backchaining implementation. The difference is, that the backchaining algorithm returns only one possible solution (in the implementation above we immediately return env), also if there are multiple solutions. Backtracking returns all possible solutions (if there are infinitely many, backtracking will not terminate).*

We can now use the backchaining algorithm to create a programming language called *Prolog*. This programming language is built on top of Horn clauses. Prolog programs are comprised of definite clauses and any question in Prolog is called a goal. *Note: We write Q :- $P_1, ..., P_n$ instead of $P_1, ..., P_n \rightarrow Q$.*

For example we can create a program, to check if a list A appended with another list B is C:

```
let appendrules =
  ["append(nil,L,L)"; "append(H::T,L,H::A) :- append(T,L,A)"];;
```

In the following example we check, if 1::2 appended with 3::4 is 1::2::3::4 using our rules for append. As we can see the first call of `append` succeeds. When we check, if 1::2 appended with 3::4 is 3::4, the backchain algorithm fails with 'tryfind' as mentioned above.

```
prolog appendrules "append(1::2::nil,3::4::nil,1::2::3::4::nil)";;
- : fol formula list = []

prolog appendrules "append(1::2::nil,3::4::nil,3::4::nil)";;
Exception: Failure "tryfind".
```

The backchain algorithm returns `env`, which contains specific ground instance that is a consequence of the clauses. So we can use the same rules, to append a list A and B into a list C. Or we can search for a list B such that A appended with B is a given list C:

```
# prolog appendrules "append(1::2::nil,3::4::nil,X)";;
- : fol formula list = [<<X = 1::2::3::4::nil>>]

prolog appendrules "append(1::2::nil,X,1::2::3::4::nil)";;
- : fol formula list = [<<X = 3::4::nil>>]
```

In this chapter we have seen a practical example, how we can use automatic proving systems. To create an efficient procedure, we reduced the set of formulas to the subset of Horn formulas. In this way we created an implementation in OCaml for the logical programming language Prolog.

9 Summary

In Section 2 we introduced the Herbrand model to reduce first-order satisfiability to propositional satisfiability. An automatic way to prove satisfiability of first-order formulas using the Herbrand model was introduced in Section 3. In the same section we introduced the Gilmore Procedure and the Davis-Putnam Procedure.

Instead of blindly trying all possibilities, we developed in Section 4 a method called unification that allows us to work with uninstantiated formulas and create instances only if we need them. A global method called tableaux which uses unification, was shown in Section 5.

We also introduced a local method in Section 6 called resolution, where variable instantiations are not propagated throughout the proof. Different improvements of this algorithm are shown in Section 7.

In the last Section 8 we reduced the set of formulas to the subset of Horn formulas. In this way we created an efficient algorithm, to proof satisfiability of Horn formulas. At the end of the report we have shown, how we can use automatic proving systems to execute Prolog programs.

References

[1] First-order tableau with unification. `http://www.wikiwand.com/en/Method_of_analytic_tableaux#/First-order_tableau_with_unification`. October 20, 2016.

[2] J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[3] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.).* Cambridge University Press, 2004.

[4] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog.* John Wiley & Sons Ltd., 1998.

A Appendix - Implementation of simple_resolution

The code provided by the book does not contain a function `simple_resolution`. So the implementation is added in this appendix:

Listing 8: Implementation of simple_resolution

```
    #use "init.ml";;

(* ------------------------------------------------------------------------- *)
(* Define resolution without subsumption                                     *)
(* ------------------------------------------------------------------------- *)
 let rec simple_resloop (used,unused) =
  match unused with
    [] -> failwith "No proof found"
  | cl::ros ->
      print_string(string_of_int(length used) ^ " used; "^
                   string_of_int(length unused) ^ " unused.");
      print_newline();
      let used' = insert cl used in
      let news = itlist(@) (mapfilter (resolve_clauses cl) used') [] in
      if mem [] news then true else simple_resloop (used',ros@news);;

 let simple_pure_resolution fm = simple_resloop([],simpcnf(specialize(pnf fm)));;

 let simple_resolution fm =
  let fm1 = askolemize(Not(generalize fm)) in
  map (simple_pure_resolution ** list_conj) (simpdnf fm1);;
```