Hauptseminar Mathematische Logik, 1. Vortrag: Decidability

by Peter Koepke

23 April 2020

1 Herbrand's Theorem

Recall:

Theorem 1. (Herbrand's Theorem) *Let S be a language which contains at least one constant symbol. Let*

$$\varphi = \forall x_0 \, \forall x_1 \ldots \forall x_{m-1} \, \psi$$

be a universal S-sentence with quantifier-free matrix ψ. Then φ is inconsistent iff there are variable-free S-terms ("constant terms")

$$t_0^0, \ldots, t_{m-1}^0, \ldots, t_0^{N-1}, \ldots, t_{m-1}^{N-1}$$

such that

$$\varphi' = \bigwedge_{i<N} \psi \frac{t_0^i \ldots t_{m-1}^i}{x_0 \ldots x_{m-1}} = \psi \frac{t_0^0 \ldots t_{m-1}^0}{x_0 \ldots x_{m-1}} \wedge \ldots \wedge \psi \frac{t_0^{N-1} \ldots t_{m-1}^{N-1}}{x_0 \ldots x_{m-1}}$$

is inconsistent.

This yields a general algorithm for proof search: to check whether $\Omega \vdash \chi$:

1. Form $\Phi = \Omega \cup \{\neg\chi\}$ and let $\varphi = \forall(\bigwedge \Phi)$ be the universal closure of $\bigwedge \Phi$. Then $\Omega \vdash \chi$ iff $\Phi = \Omega \cup \{\neg\chi\}$ is inconsistent iff $(\bigwedge \Phi) \vdash \bot$ iff $\forall(\bigwedge \Phi) \vdash \bot$.

2. Transform φ into universal form $\varphi^{\forall} = \forall x_0 \forall x_1 \ldots \forall x_{m-1}\, \psi$ (SKOLEMization).

3. (Systematically) search for constant S-terms

$$t_0^0, \ldots, t_{m-1}^0, \ldots, t_0^{N-1}, \ldots, t_{m-1}^{N-1}$$

   such that

$$\varphi' = \bigwedge_{i<N} \psi \frac{t_0^i \ldots t_{m-1}^i}{x_0 \ldots x_{m-1}} = \psi \frac{t_0^0 \ldots t_{m-1}^0}{x_0 \ldots x_{m-1}} \wedge \ldots \wedge \psi \frac{t_0^{N-1} \ldots t_{m-1}^{N-1}}{x_0 \ldots x_{m-1}}$$

   is inconsistent.

4. If an inconsistent φ' is found, output "yes", otherwise carry on.

2 Implementing Herbrand's Theorem in OCaml

Here is the implementation of the proof method from Harrison's Handbook:

```
let gilmore fm =
  let sfm = skolemize(Not(generalize fm)) in
```

```
let fvs = fv sfm and consts,funcs = herbfuns sfm in
let cntms = image (fun (c,_) -> Fn(c,[])) consts in
length(gilmore_loop (simpdnf sfm) cntms funcs fvs 0 [[]] [] []);;
```

`gilmore` is a function, composed of other functions whose definition is also in Harrison's handbook. We can try `gilmore` in `OCaml`, by starting the REPL with the files from Harrison's book. This is organized by a Makefile:

```
koepke@dell:~/Desktop/OCaml$ make

# gilmore <<exists y. forall x. P(y) ==> P(x)>>;;
0 ground instances tried; 1 items in list
0 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
- : int = 2

#
```

Some information on `gilmore`:

1. `gilmore` expects a formula from the data type `formula`:

```
type ('a)formula = False
                 |True
                 |Atom of 'a
                 |Not of ('a)formula
                 |And of ('a)formula * ('a)formula
                 |Or of ('a)formula * ('a)formula
                 |Imp of ('a)formula * ('a)formula
```

```
                  |Iff of ('a)formula * ('a)formula
                  |Forall of string * ('a)formula
                  |Exists of string * ('a)formula;;
```

This type has a *type variable* `'a` for the atomic formulas. In our case that should be

```
type fol = R of string * term list;;
```

where

```
type term = Var of string

          | Fn of string * term list;;
```

For example, $x+y<z$ can be formalized as the atomic formula:

```
Atom(R("<",[Fn("+",[Var "x"; Var "y"]); Var "z"]))
```

The functions involved in `gilmore` work in the data type `fol formula`:

```
let generalize fm = itlist mk_forall (fv fm) fm;;
```

the function `fv` makes a list of the free variables of a formula; it is defined recursively in the familiar way:

```
let rec fv fm =
  match fm with
    False | True -> []
  | Atom(R(p,args)) -> unions (map fvt args)
  | Not(p) -> fv p
  | And(p,q) | Or(p,q) | Imp(p,q) | Iff(p,q) -> union (fv p) (fv q)
  | Forall(x,p) | Exists(x,p) -> subtract (fv p) [x];;
```

`mk_forall` is the simple operation of putting a universally quantified variable in front of a formula:

```
let mk_and p q = And(p,q) and mk_or p q = Or(p,q)
and mk_imp p q = Imp(p,q) and mk_iff p q = Iff(p,q)
and mk_forall x p = Forall(x,p) and mk_exists x p = Exists(x,p);;
```

`itlist` is a "utility function" that lifts an operation to a list of variables:

```
itlist f [1;2;3] x = f 1 (f 2 (f 3 x))
```

We can get some information about "objects" in OCaml from their type. In the terminal:

```
# Atom(R("<",[Fn("+",[Var "x"; Var "y"]); Var "z"]));;
- : fol formula = <<x + y < z>>
#

# fv (Atom(R("<",[Fn("+",[Var "x"; Var "y"]); Var "z"])));;
- : string list = ["x"; "y"; "z"]
#
```

We can use "pretty parsing" and "pretty printing" of formulas:

```
# fv <<x + y < z>>;;
- : string list = ["x"; "y"; "z"]
#
```

And then the generalization of that formula is:

```
# generalize <<x + y < z>>;;
```

```
- : fol formula = <<forall x y z. x + y < z>>
#
```

3 The Decision Problems

`gilmore` is a complete proof procedure for first-order logic: if φ is universally valid (in all models) then `gilmore phi` will (in principle) stop after a finite time and has found a proof (a Herbrand-style inconsistency).

`gilmore` is a semi-decision procedure:

- if φ is valid, `gilmore` will certify this after a finite time;
- if φ is not valid, `gilmore` will run forever.

The formula $\exists x\, P(x)$ is not universally valid:

```
# gilmore << exists x. P(x) >>;;
```

leads into an infinite loop:

```
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
```

```
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
^C1 ground instances tried; 1 items in listInterrupted.
#
```

The converse $\neg\exists x\, P(x)$ is not universally valid either:

```
# gilmore << exists x. P(x) >>;;
```

similarly leads to:

```
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
1 ground instances tried; 1 items in list
^C1 ground instances tried; 1 items in list
```

```
Interrupted.
#
```

So `gilmore` techniques are not able to *decide* the validity of $\exists x\, P(x)$. (We know that $\exists x\, P(x)$ and $\neg\exists x\, P(x)$ are both consistent, since there are interpretations of P which make both true.)

What we need is a

> (3) Test whether a formula is valid or invalid (or whether it is satisfiable or unsatisfiable). [copied from Harrison]

This is Hilbert's *Entscheidungsproblem*. Turing has given a negative answer to this in his article:

> Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society (2), 42, 230–265.

There is no procedure or algorithm solving the Entscheidungsproblem.

So we can only hope for solutions of restricted Entscheidungsprobleme. We look at the following restriction: T is an (interesting) theory or axiom system; is the property $T \vdash \varphi$ decidable?

Definition 2. *An L-theory T is* decidable *if there is an algorithm P such that for every L-sentence φ the algorithm P with input $\ulcorner\varphi\urcorner$ halts and outputs*

- *"yes" iff $P \vdash \varphi$;*
- *"no" iff $P \nvdash \varphi$.*

This models the standard problem in theoretical mathematics: work in a fixed basic theory T(like Peano arithmetic, field theory, or set theory) and decide whether φ is a consequence of T.

Definition 3. *An L-theory T is* complete *if T is consistent, and for every L-sentence φ*

$$T \vdash \varphi \textit{ or } T \vdash \neg\varphi.$$

Theorem 4. *If T is finitely axiomatizable and complete, then T is decidable.*

Proof. Let P be the following algorithms: after inputting the L-sentence φ:

- run `gilmore` $(\bigwedge T \rightarrow \varphi)$ and `gilmore` $(\bigwedge T \rightarrow \neg\varphi)$ in parallel;
- since T is consistent and complete, exactly one of these processes terminates;
- if `gilmore` $(\bigwedge T \rightarrow \varphi)$ terminates, output "yes";
- if `gilmore` $(\bigwedge T \rightarrow \neg\varphi)$ terminates, output "no".

Since $T \nvdash \varphi$ is equivalent to $(\bigwedge T \rightarrow \neg\varphi)$ this is a decision algorithm. □

Note: Harrison says:

> this is usually not a very practical approach, so we will focus on more direct methods of proving decidability.

Some important theories are complete and hence decidable:

- Dense linear orders without endpoints (DLO);
- algebraically closed fields in a given characteristic, e.g., the theory of $\mathbb{C}$;
- ...

Other theories are incomplete:

- group theory;
- field theory;
- ...

The Gödel incompleteness theorems yields many interesting incomplete theories:

Theorem 5. *If* ST *is consistent then* ST *is incomplete, i.e., there is an* $\in$*-sentence* φ *such that* $\mathrm{ST} \nvdash \varphi$ *and* $\mathrm{ST} \nvdash \neg\varphi$.

ZFC (i.e., "mathematics") is incomplete.

Are (some of) the incomplete theories nevertheless decidable?

Are there efficient algorithms?

4 Quantifier Elimination

Definition 6. *A theory T in a first-order language L admits quantifier elimination if for each formula p of L, there is a quantifier-free formula q with $FV(q) \subseteq FV(p)$ such that $T \models p \Leftrightarrow q$ (or as we sometimes say, p and q are T-equivalent). As usual, we are interested in constructing quantifier-free equivalents by an algorithmic process, rather than merely showing that they exist in principle.*

Quantifier elemimation may lead to decidability:

For an L-sentence p let q be a T-equivalent quantifier-free L-sentence. Often quantifier-free sentences can be decided by calculation. Quantifier-free sentences in the theory of fields are boolean combinations of equalities between terms built from the constants 0 and 1 by $+$ and $*$. Typical such equalities are $(1+1)*(1+1) = 1+1+1+1$ (true) or $(1+1)*(1+1) = 1+1+1$ (false) (with some bracketing).

Harrison reduces the general question of quantifier elimination to a restricted one:

> Quite generally, to establish quantifier elimination for arbitrary first-order formulas, it suffices to demonstrate it for formulas with the following rather special form: $\exists x \bigvee \bigwedge \text{Literale} \leftrightarrow \bigvee \exists x \bigwedge \text{Literale}$
> ∃x. α 1 ∧ . . . ∧ α n
> with each α i a literal (either an atomic formula or the negation of an atomic formula) containing x. The basic idea is that we can apply this elimination

successively from the innermost quantifier to the outermost, transforming $\forall$x.P [x] into $\neg$($\exists$x.$\neg$P [x]) and always putting the body in disjunctive normal form and distributing the existential quantifier over it.

So if we think of

- `afn` as an auxiliary function that equivalently transforms atomic formulas; e.g., $x \leqslant y \mapsto \neg y < x$, if we consider dense linear orders;

- `nfn` as the transformation to disjunctive normal form;

- `qfn` as the single quantifier elimination procedure for formulas of the form $\exists$x. α 1 ∧ ... ∧ α n

then the idea above of lifting `qfn` to arbitrary formulas `fm` is incorporated into

```
let lift_qelim afn nfn qfn =
  let rec qelift vars fm =
    match fm with
    | Atom(R(_,_)) -> afn vars fm
    | Not(p) -> Not(qelift vars p)
    | And(p,q) -> And(qelift vars p,qelift vars q)
    | Or(p,q) -> Or(qelift vars p,qelift vars q)
    | Imp(p,q) -> Imp(qelift vars p,qelift vars q)
    | Iff(p,q) -> Iff(qelift vars p,qelift vars q)
    | Forall(x,p) -> Not(qelift vars (Exists(x,Not p)))
    | Exists(x,p) ->
          let djs = disjuncts(nfn(qelift (x::vars) p)) in
```

```
          list_disj(map (qelim (qfn vars) x) djs)
      | _ -> fm in
    fun fm -> simplify(qelift (fv fm) (miniscope fm));;
```

We shall later encounter that lift for specific theories, e.g.,

```
  let complex_qelim =
    simplify ** evalc **
    lift_qelim polyatom (dnf ** cnnf (fun x -> x) ** evalc)
               basic_complex_qelim;;
```

5 Dense linear orders

Dense linear orders were axiomatized by the finite axiom system DLO:

$$\begin{aligned}
&\forall x y. x{=}y \vee x{<}y \vee y{<}x,\\
&\forall x y z. x{<}y \wedge y{<}z \Rightarrow x{<}z,\\
&\forall x. \neg(x{<}x),\\
&\forall x y. x{<}y \Rightarrow \exists z. x{<}z \wedge z{<}y,\\
&\forall x. \exists y. x{<}y,\\
&\forall x. \exists y. y{<}x.
\end{aligned}$$

There quantifier elimination is defined by:

```
  let quelim_dlo =
    lift_qelim afn_dlo (dnf ** cnnf lfn_dlo) (fun v -> dlobasic);;
```

where

```
let lfn_dlo fm =
  match fm with
    Not(Atom(R("<",[s;t]))) -> Or(Atom(R("=",[s;t])),Atom(R("<",[t;s])))
  | Not(Atom(R("=",[s;t]))) -> Or(Atom(R("<",[s;t])),Atom(R("<",[t;s])))
  | _ -> fm;;
```

and

```
let dlobasic fm =
  match fm with
    Exists(x,p) ->
      let cjs = subtract (conjuncts p) [Atom(R("=",[Var x;Var x]))] in
      try let eqn = find is_eq cjs in
          let s,t = dest_eq eqn in
          let y = if s = Var x then t else s in
          list_conj(map (subst (x |=> y)) (subtract cjs [eqn]))
      with Failure _ ->
          if mem (Atom(R("<",[Var x;Var x]))) cjs then False else
          let lefts,rights =
            partition (fun (Atom(R("<",[s;t]))) -> t = Var x) cjs in
          let ls = map (fun (Atom(R("<",[l;_]))) -> l) lefts
          and rs = map (fun (Atom(R("<",[_;r]))) -> r) rights in
          list_conj(allpairs (fun l r -> Atom(R("<",[l;r]))) ls rs)
  | _ -> failwith "dlobasic";;
```

`dlobasic` φ requires, that φ is of the form $\exists x\,\psi$[moreover, ψ will be a conjunction of (positive) literals that all contain the variable x].

The list `cjs` of relevant literals is formed by removing the tautological formula $x \equiv x$ from the list of conjuncts of ψ.

The program tries to find an equation `eqn` in `cjs`. This equation is of the form $x \equiv y$ or $y \equiv x$ [where y is a variable distinct from x].

Then the result is formed by removing `eqn` from the conjuncts `cjs`, the substitution of x by y in the remaining conjects, and subsequent conjunction. This uses the equivalence

$$\exists x(x \equiv y \wedge \phi(x)) \leftrightarrow \phi(y),$$

with a quantifier-free right hand side.

If there is no such equation `eqn` then ψ consists only of inequalities of the form $u < x$ or $x < v$.

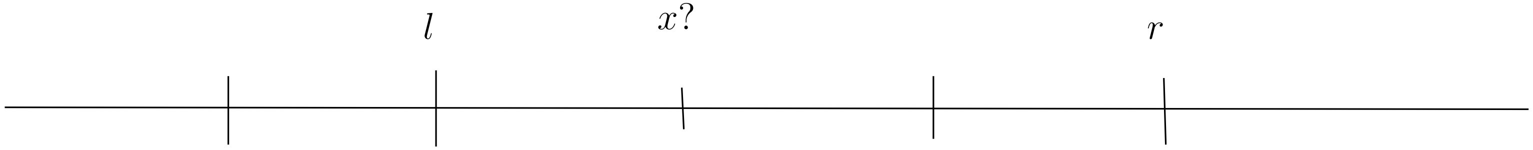
If $x < x$ is amongst them, the result is $\perp$ (`False`) because of the equivalence

$$\mathrm{DLO} \vdash \exists x(x < x \wedge \phi) \leftrightarrow \perp$$

Otherwise, the inequalities are of the forms $u < x$ or $x < v$ with variables $u, v \neq x$. Then

$$\mathrm{DLO} \vdash \exists x(u_0 < x \wedge \ldots \wedge u_{m-1} < x \wedge x < v_0 \wedge \ldots \wedge x < v_{n-1}) \leftrightarrow \bigwedge_{i<m, j<n} u_i < v_j$$

and the quantifier-free right-hand side is the result.


l
$x?$
r