Review

- More OCaml
- Stålmarck’s Method
- Binary Decision Diagrams
Outline

- Partial Functions in OCaml
- First Order Logic in OCaml
- Prenex Normal Form and Skolemization
- Herbrand’s Theorem and Gilmore’s Procedure
- Unification

Sources:
Harrison, John. *Introduction to Logic and Automated Theorem Proving*. Unpublished manuscript. Used by permission.

Partial Functions in OCaml

Harrison’s OCaml library (in *lib.ml*) provides a convenient interface for building and using partial functions.

We will go through some examples to explain how it works. The examples are in *pf.ml*.

Note that from now on, the operator \( := \) builds a trivial (defined at one point) partial function. For updating a reference variable, use *refassign*. 

First Order Logic

The OCaml code for reasoning about first order logic is in `fol.ml`

It includes:

- Types for terms and formulas of first order logic
- Parser and printer for terms and formulas
- Code for evaluating in a finite interpretation (model)
- Code for safe substitution

We will go through it briefly.
Prenex Normal Form

A prenex formula is one of the form $Q_1x_1 \cdots Q_nx_n\alpha$, where each $Q_i$ is a quantifier and $\alpha$ is quantifier-free.

For any formula we can find a logically equivalent prenex formula as follows.

- Simplify away $false$, $true$, and vacuous quantification.
- Eliminate implication and equivalence, and push down negation.
- Pull out quantifiers.

For the last step, we use the usual identities for pulling quantifiers out of conjunctions and disjunctions, and take care to avoid variable conflicts. For example:

$$(\alpha \land \exists x \beta) \leftrightarrow \exists y (\alpha \land [x := y]\beta), \text{ where } y \text{ is a new variable.}$$

However, we can also be a bit more clever when pulling out conjunctions of universal, or disjunctions of existential quantifiers:

$$(\forall x. p) \land (\forall y. q) \leftrightarrow \forall z. ([x := z]p \land ([y := z]q)$$

$$(\exists x. p) \lor (\exists y. q) \leftrightarrow \exists z. ([x := z]p \lor ([y := z]q)$$
Prenex Normal Form

Example

Find an equivalent in prenex form:

$$\exists y_1. \forall x_1. \ (\forall y_2. \exists x_2. \ x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. \ y_1 + x_3 > y_3)$$
Prenex Normal Form

Example

Find an equivalent in prenex form:

$$\exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1. [(\exists y_2. \forall x_2. \neg (x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$
Prenex Normal Form

Example

Find an equivalent in prenex form:

$$\exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1. [(\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1 x_3. [(\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\exists y_3. y_1 + x_3 > y_3)]$$
Prenex Normal Form

Example

Find an equivalent in prenex form:

$$\exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1. [(\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1 x_3. [(\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1 x_3. \exists y_2. [(\forall x_2. \neg(x_1 + y_2 < x_2)) \lor (y_1 + x_3 > y_2)]$$
Prenex Normal Form

Example

Find an equivalent in prenex form:

\[ \exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

eff

\[ \exists y_1. \forall x_1. [(\exists y_2. \forall x_2. \neg (x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

eff

\[ \exists y_1. \forall x_1 x_3. [(\exists y_2. \forall x_2. \neg (x_1 + y_2 < x_2)) \lor (\exists y_3. y_1 + x_3 > y_3)] \]

eff

\[ \exists y_1. \forall x_1 x_3. \exists y_2. [(\forall x_2. \neg (x_1 + y_2 < x_2)) \lor (y_1 + x_3 > y_2)] \]

eff

\[ \exists y_1. \forall x_1 x_3. \exists y_2. \forall x_2. [\neg (x_1 + y_2 < x_2)) \lor (y_1 + x_3 > y_2)] \]
Skolemization

For any first-order formula, there is a logically equivalent second-order formula consisting of:

1. First a string (possibly empty) of existential individual and function quantifiers, followed by
2. A string (possibly empty) of universal individual quantifiers, followed by
3. A quantifier-free formula

First, given any first order formula, we can find an equivalent formula $\phi$ in prenex normal form. If $\phi$ is in the form described above, we are done.

Otherwise, $\phi$ must contain a sub-formula of the form $\forall v_1 \ldots \forall v_n \exists z \psi(z)$.

This formula is equivalent to $\exists F \forall v_1 \ldots \forall v_n \psi(Fv_1 \ldots v_n)$.

By repeatedly applying this rule (from the outside inwards), every nested existential quantifier can be eliminated. The result is a formula of the form described above.

In practice, we don’t need to actually convert to prenex normal form. We can just skolemize “in-place” as we recursively traverse the formula.
Skolemization

Example

Skolemize:

$$\exists y_1. \forall x_1. \left((\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)\right)$$
Skolemization

Example

Skolemize:

$$\exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$

iff

$$\exists y_1. \forall x_1. [(\exists y_2. \forall x_2. -(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)]$$
**Skolemization**

**Example**

Skolemize:

\[ \exists y_1. \forall x_1. \left[ (\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right] \]

iff

\[ \exists y_1. \forall x_1. \left[ (\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right] \]

iff

\[ \exists y_1. \exists F_{y_2}. \forall x_1. \left[ (\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right] \]
**Skolemization**

**Example**

Skolemize:

\[ \exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \forall x_1. [(\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \exists F_{y_2}. \forall x_1. [(\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \exists F_{y_2}. \exists F_{y_3}. \forall x_1. [(\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (\forall x_3. y_1 + x_3 > F_{y_3}(x_3))] \]
Skolemization

Example

Skolemize:

\[ \exists y_1. \forall x_1. [(\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \forall x_1. [\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)] \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \exists F_{y_2}. \forall x_1. [\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)] \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3)] \]

iff

\[ \exists y_1. \exists F_{y_2}. \exists F_{y_3}. \forall x_1. [\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)] \lor (\forall x_3. y_1 + x_3 > F_{y_3}(x_3)] \]

iff

\[ \exists y_1. \exists F_{y_2}. \exists F_{y_3}. \forall x_1 x_2 x_3. [\neg(x_1 + F_{y_2}(x_1) < x_2)] \lor (y_1 + x_3 > F_{y_3}(x_3)] \]
Skolemization

Example

Skolemize:

\[ \exists y_1. \forall x_1. \left( (\forall y_2. \exists x_2. x_1 + y_2 < x_2) \rightarrow (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right) \]

iff

\[ \exists y_1. \forall x_1. \left( (\exists y_2. \forall x_2. \neg(x_1 + y_2 < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right) \]

iff

\[ \exists y_1. \exists F_{y_2}. \forall x_1. \left( (\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (\forall x_3. \exists y_3. y_1 + x_3 > y_3) \right) \]

iff

\[ \exists y_1. \exists F_{y_2}. \exists F_{y_3}. \forall x_1. \left( (\forall x_2. \neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (\forall x_3. y_1 + x_3 > F_{y_3}(x_3)) \right) \]

iff

\[ \exists y_1. \exists F_{y_2}. \exists F_{y_3}. \forall x_1 x_2 x_3. \left( (\neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (y_1 + x_3 > F_{y_3}(x_3)) \right) \]

which is equisatisfiable with

\[ \forall x_1 x_2 x_3. \left( (\neg(x_1 + F_{y_2}(x_1) < x_2)) \lor (y_1 + x_3 > F_{y_3}(x_3)) \right) \]
Skolemization

Harrison defines first order satisfiability as follows:

A formula $\phi$ is satisfiable if there exists a model $M$ such that for all variable assignments $s$, $\models_M \phi[s]$.

This definition is useful in the context of skolemization as it allows us to get rid of all quantifiers.

Thus, skolemization can be used to construct a quantifier-free equisatisfiable (according to the above definition) formula.

The algorithms we consider next require a skolemized formula. The OCaml code for prenex normal form and skolemization is in `skolem.ml`. 
Herbrand’s Theorem

Skolemization reduces the problem of first order satisfiability to first order satisfiability of a quantifier-free formula.

Our goal is to further reduce this to just propositional satisfiability.

Notice that because of the way we defined propositional and first order formulas, every quantifier-free first order formula can be thought of as a propositional formula in which the atomic formulas are the propositional variables.

This brings us to a somewhat surprising result.
Herbrand’s Theorem

**Theorem**

A quantifier-free first order formula is a propositional tautology if and only if it is first order valid.

**Proof**

It is easy to see that a propositional tautology must be first order valid.

In the other direction, consider the model which maps each term to itself (also called a *canonical* or *term* model).

In such a model, all syntactically different terms map to different elements of the domain.

Thus, we can choose the value to which each syntactically different atomic formula evaluates independently.

In this way, we can define the meaning of each predicate symbol so that the value of the atomic formulas coincide with a given propositional variable assignment.

Thus, a formula which is first order valid must be propositionally true for every possible propositional variable assignment.
Herbrand’s Theorem

For satisfiability, one direction is the same:

**Corollary**

If a quantifier-free first order formula is first order satisfiable, it is also propositionally satisfiable.

However, the opposite direction does not hold. What does hold is a weaker result known as *Herbrand’s Theorem*. First we need a couple of definitions.

A *Herbrand interpretation* for $p$ is a canonical model with minimal domain, i.e. consisting only of terms built from the function and constant symbols in $p$.

If $p$ contains no constant symbols, we choose one, say $c_0$, arbitrarily, so that the domain will not be empty.

Given a formula $p$ and a propositional valuation $d$ over the atomic formulas of the Herbrand interpretation for $p$, we denote the Herbrand interpretation for $p$ whose predicate symbols are interpreted to coincide with $d$ as $H_{p,d}$.

Given a Herbrand interpretation, a *ground instance* of a formula $p$ is the result of replacing every free variable with some element of the Herbrand domain for $p$. 
**Herbrand’s Theorem**

**Herbrand’s Theorem**

A quantifier-free formula $p$ is first order satisfiable iff the set of all ground instances is (simultaneously) propositionally satisfiable.

**Proof**

Suppose $p$ is first order satisfiable. Then it holds in some model $M$ under all variable assignments. Now, let $q$ be some ground instance of $p$. Consider the propositional valuation $d$ of atomic formulas in $q$ given by the first order evaluation under $M$. Because $M$ satisfies $p$ under all variable assignments, it must be the case that $d$ propositionally satisfies $q$.

Conversely, suppose $d$ is a propositional valuation satisfying all ground instances of $p$. Then $H_{p,d}$ is a model satisfying $p$ for all variable assignments. This is because the value under a variable assignment $s$ in the model $H_{p,d}$ corresponds to the value of the ground instance formed by applying $s$ as a substitution to $p$. □
Herbrand’s Theorem

Now, using the compactness theorem for first order logic, we obtain the following.

**Theorem**

A quantifier-free formula is first order satisfiable iff all finite sets of ground instances are propositionally satisfiable.

**Corollary**

A quantifier-free formula $p$ is first order unsatisfiable iff some finite set of ground instances of $p$ is propositionally unsatisfiable.

This leads naturally to a procedure for checking the validity of a formula $p$. We simply try to show that its negation is unsatisfiable by enumerating larger and larger sets of ground instances and testing them for propositional satisfiability.

If we guarantee that all finite sets of ground instances are eventually tried, then this gives us a semi-decision procedure for validity of first order formulas.

Of course, the undecidability of first order logic means we cannot do better.
Gilmore’s Procedure

One of the earliest implementations of a proof system based on enumerating larger and larger sets of ground instances was due to Gilmore in 1960.

We will look at a simple implementation of Gilmore’s procedure as well as several variations in herbrand.ml.
Unification

The last few examples in *herbrand.ml* demonstrate that often, relatively few instances are needed to show unsatisfiability.

The problem becomes finding an intelligent way to find these few instances, rather than searching blindly through all the instances in the Herbrand universe.

The idea behind *unification* is to make intelligent choices about how to instantiate variables.
Unification

Given a set of pairs of terms, \( S = \{(s_1, t_1), \ldots, (s_n, t_n)\} \), a unifier of the set \( S \) is an instantiation \( \sigma \) such that

\[
\text{termsubst} \sigma s_i = \text{termsubst} \sigma t_i
\]

for each \( i = 1, \ldots, n \).

Unifying a set of pairs is analogous to solving a system of simultaneous equations.

The unification problem may be unsolvable. For example, there is no unifier for \( f(x) \) and \( g(y) \), because the instantiated terms will always differ at the top level.

Similarly, there is no unifier of \( x \) and \( f(x) \), or of \( x \) and any term containing \( x \) as a proper subterm. This is like trying to solve the equation \( x = x + 1 \).

A more complicated example of circularity is \( \{(x, f(y)), (y, g(x))\} \), which is like solving the equations \( x = y + 1 \) and \( y = x + 2 \).
Unification

We now give a general unification algorithm which proceeds by a sequence of transformations, each of which converts a problem $S$ to a new problem $S'$ such that $S$ and $S'$ have exactly the same set of unifiers.

Suppose we have the unification problem

$$\{(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n))\} \cup S.$$

This has exactly the same unifiers as

$$\{(s_1, t_1), \ldots (s_n, t_n)\} \cup S,$$

because any instantiation unifies $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$ iff it unifies each corresponding pair $s_i$ and $t_i$.

By using this transformation repeatedly, we can either recognize unsolvability (when a pair of terms has different top-level functions) or reduce the unification problem to a new one in which at least one element of each pair is a variable.
Unification

A simplified unification problem may still be unsolvable because of cycles.

The strategy we will adopt is to gradually convert a set $eqs$ of pairs of terms into a finite partial function, $env$, from variables to terms, checking for cycles along the way.

To be more precise, we define $x \longrightarrow y$ to mean that $env(x) = t$ with $y$ a free variable of $t$.

Now, to add a new pair $(x, t)$ to $env$, we first ensure that $env$ does not already have a mapping for $x$. Second, we ensure that there is no free variable $y$ in $t$ such that $y \longrightarrow^* x$. If these conditions are satisfied, then we can update $env$ with $(x, t)$ and it will still be cycle-free.

There is one special case in which a cycle is allowed. This is the case in which we have $x_0 \longrightarrow \cdots \longrightarrow x_p \longrightarrow x_0$ and for each $x_i \longrightarrow x_j$, we have not just $x_j$ is a free variable of $env(x_i)$, but in fact $x_j = env(x_i)$.

In this case, we can safely ignore the cycle (as well as ignoring the new pair $(x, t)$ since it is already captured by $env$).

The OCaml code for unification is in $unif.ml$. 