2.15 *Extended example:* Verifying arguments automatically

The ML programming language was originally developed for writing and extending software for automatic theorem proving. When you observe the formal, almost mechanical process we have used in the games of this chapter, you can see why people would reason that if a computer can do arithmetic, it should also be able to produce or at least verify arguments.

In this section, we will build an ML program to play Game 2, or the equivalent—that is, a program that will determine whether or not an argument is valid. In our program, however, we will approach this verification at a different angle from what we used in Game 2.

Recall the alternative definition of a valid argument. If $p_1, p_2, \ldots p_n$ are premises and s is a conclusion, then the argument is valid if $(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow s$ is a tautology. For example, if you took the truth table you made for verifying division into cases in Exercise 2.8.5 and added an extra column for

$$((p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow r$$

you would find that that column is true for every assignment to p, q, and r. For our automatic argument checker, we will convert the argument into a formula like this and then check to see that it is a tautology.

First, we need a way to represent logical formulas—we will simply call them *propositions*—in ML. The propositions we have used can be

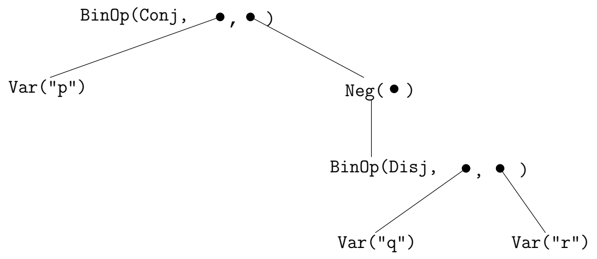
- Variables, say p, q, r, ...
- Negations of propositions, say $\sim p$ or $\sim (q \vee (p \wedge r))$.
- Conjunctions of propositions, say $(p \vee q) \wedge r$.
- Disjunctions of propositions, say $\sim q \vee r$.
- Conditionals of propositions, say $p \rightarrow \sim (q \vee r)$.

A proposition can contain smaller propositions—our definition of a proposition (by listing the different kinds) is self-referential. This means that if we make an ML datatype proposition, it will need to be self-referential as well. Fortunately ML datatypes, like ML functions, can be defined recursively.

To simplify things a bit, we first eliminate conditionals, because any conditional can be expressed with negation and disjunction: $p \rightarrow q \equiv \sim p \vee q$. Also, we will consider conjunctions and disjunctions to be two kinds of binary operations.

```
datatype binLogOp = Conj | Disj;
datatype proposition = Var of string
                     | Neg of proposition
                     | BinOp of binLogOp * proposition * proposition;
```

When a value of this datatype represents a proposition, for example $p \wedge \sim (q \vee r)$, it will have a branching structure like this



To make it a little easier to make conjunctions, disjunctions, and conditionals, we add a few functions:

```
fun conj(p, q) = BinOp(Conj, p , q);
fun disj(p, q) = BinOp(Disj, p, q);
fun cond(p, q) = BinOp(Disj, Neg(p), q);
```

It is always helpful to write a function that produces a string representation of a datatype. In this case, adding appropriate parentheses is the tricky part

```
fun display(Var(s)) = s
  | display(Neg(Var(s))) = "~" ^ s
  | display(Neg(p)) = "~(" ^ display(p) ^ ")"
  | display(BinOp(oper, p, q)) =
    (case p of
         BinOp(subOp, pp, qq) => if subOp = oper
                                 then display(p)
                                 else "(" ^ display(p) ^ ")"
       | _ => display(p)) ^
    (case oper of
         Conj => "^"
       | Disj => "v") ^
    (case q of
         BinOp(subOp, pp, qq) => if subOp = oper
                                 then display(q)
                                 else "(" ^ display(q) ^ ")"
       | _ => display(q));
```

Trying it out:

```
- conj(Var("p"), Neg(disj(Var("q"), Var("r"))));

val it = BinOp (Conj,Var "p",Neg (BinOp (#,#,#))) : proposition
```

```
- display(it);

val it = "p^~(qvr)" : string
```

The arguments we want to verify will have long sequences of conjunctions since we want to conjoin all our premises. To make this easier, we write a function that will turn a list of propositions into a proposition that is the equivalent of anding all those propositions together. While we are at it, we may as well do the same for disjunctions.

```
exception EmptyBigConjOrDisj;

fun bigConj([]) = raise EmptyBigConjOrDisj
  | bigConj([p]) = p
  | bigConj(p::rest) = conj(p, bigConj(rest));

fun bigDisj([]) = raise EmptyBigConjOrDisj
  | bigDisj([p]) = p
  | bigDisj(p::rest) = disj(p, bigDisj(rest));
```

We can now use this to encode an argument. Take the argument in Exercise 2.9.1

(a) $t \to u$
(b) $p \vee \sim q$
(c) $p \to (u \to r)$
(d) q
(e) $\therefore t \to r$

```
- val orig = cond(bigConj([cond(Var("t"), Var("u")),
=                   BinOp(Disj, Var("p"), Neg(Var("q"))),
=                   cond(Var("t"), cond(Var("u"), Var("r"))),
=                   Var("q")]),
=        cond(Var("t"), Var("r")));

val orig = BinOp (Disj,Neg (BinOp (#,#,#)),BinOp (Disj,Neg #,Var #))
  : proposition

-  print(display(orig) ^ "\n");

~((~tvu)^(pv~q)^(~tv~uvr)^q)v~tvr
val it = () : unit
```

Now for the real work. We have a formula equivalent to our argument, and we want to check that it is true for any assignment to the variables. We could take a "brute force" approach, equivalent to making a truth table: enumerate every possible combination of true and false values and evaluate the formula for each one of them. While this would make an interesting exercise (see Project 2.15.C), there is a more efficient way.

Consider the proposition

$$(p \vee q \vee r) \wedge (q \vee \sim t \vee \sim u) \wedge (u \vee \sim p \vee t)$$

This is not a tautology—just pick all of p, q, and r to be false. That makes the first subproposition $(p \vee q \vee r)$ false, and since we are joining several propositions together with conjunctions, that makes the entire proposition false. On the other hand,

$$(p \vee q \vee r \vee \sim q) \wedge (q \vee \sim t \vee \sim u \vee t) \wedge (u \vee \sim p \vee t \vee p)$$

is a tautology, and this is how we tell: since we again are anding several subpropositions together, this entire proposition will be true exactly under those situations where every subproposition is true. So, this proposition is a tautology (always true) if and only if all of its subpropositions are tautologies.

The first subproposition is always true because no matter how you assign the variables, either q or $\sim q$ must be true, and either of them will make the whole subproposition true. If a chain of disjunctions contains both a variable and its negation, then that chain of disjunctions is a tautologies. The other two subpropositions in this example are tautologies because they contain both $\sim t$ and t and both $\sim p$ and p.

We picked an easy one, however: If the proposition is a big conjunction of subpropositions, each of which is a big disjunction of variables or negations of variables, then all we need to do is see if each subproposition contains some variable and its negation. But this is the core of our strategy: To determine if something is a tautology, we will first transform it into an equivalent "easy one," and then use a simple test to see if it is a tautology.

We will do this in two steps. First, we want to make sure that negations are applied only to variables—not to binary operations, not to other negations. A proposition is in *negation normal form* if the only negations in it are applied directly to variables. Transforming a proposition to negation normal form is straightforward: work from the outside, top-level proposition into the subpropositions; if you find a double negation, remove both because they cancel each other out; if the negation is applied to a binary operation, then push it in and flip the operator, following De Morgan's laws. For example,

negation normal form

$$\begin{aligned} & \sim ((p \vee \sim q) \wedge \sim r) \\ \equiv \quad & \sim (p \vee \sim q) \vee \sim\sim r \\ \equiv \quad & (\sim p \wedge \sim\sim q) \vee r \\ \equiv \quad & (\sim p \wedge q) \vee r \end{aligned}$$

Writing a function for this is intuitive when you use pattern matching. Specifically, the interesting cases are the various things a negation can be applied to:

```
fun flip(Conj) = Disj
  | flip(Disj) = Conj;

fun negNormForm(Var(s)) = Var(s)
  | negNormForm(Neg(Var(s))) = Neg(Var(s))
  | negNormForm(Neg(Neg(p))) = negNormForm(p)
  | negNormForm(Neg(BinOp(oper, p, q))) =
    BinOp(flip(oper), negNormForm(Neg(p)), negNormForm(Neg(q)))
  | negNormForm(BinOp(oper, p, q)) =
    BinOp(oper, negNormForm(p), negNormForm(q));
```

Try it:

```
- val nnf = negNormForm(orig);

val nnf = BinOp (Disj,BinOp (Disj,BinOp #,BinOp #),
                 BinOp (Disj,Neg #,Var #))  : proposition

- print(display(nnf) ^ "\n");

(t^~u)v(~p^q)v(t^u^~r)v~qv~tvr
val it = () : unit
```

What we had been calling "easy ones" are actually propositions in , which means that the proposition is the conjunction of subpropositions, each of which is the disjunction of variables and negations of variables. Just as the transformation to negation normal form uses De Morgan's law, the transformation to conjunctive normal form uses the distributive law, specifically $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$. For example,

conjunctive normal form

$$\begin{aligned} & \sim q \vee (r \vee (\sim p \wedge q)) \\ \equiv \quad & \sim q \vee ((r \vee \sim p) \wedge (r \vee q)) \\ \equiv \quad & (\sim q \vee r \vee \sim p) \wedge (\sim q \vee r \vee q) \end{aligned}$$

The code for conversion to conjunctive normal form is harder to follow. We start with a function that takes two propositions and distributes the first over the second using disjunction. This is only interesting if the second proposition is a conjunction—otherwise we merely create a disjunction. The function for converting to conjunctive normal form assumes the proposition is already in negation normal form and raises an exception if it is not. The interesting case is if the position being transformed is a conjunction: then we transform the two subproposition and distribute.

```
fun distribute(p, BinOp(Conj, q, r)) =
        BinOp(Conj, distribute(p, q), distribute(p, r))
  | distribute(BinOp(Conj, p, q), r) =
        BinOp(Conj, distribute(p, r), distribute(q, r))
  | distribute(p, q) = BinOp(Disj, p, q);

exception NotInNNF of string;

fun conjNormForm(Var(s)) = Var(s)
  | conjNormForm(Neg(Var(s))) = Neg(Var(s))
  | conjNormForm(Neg(p)) = raise NotInNNF(display(Neg(p)))
  | conjNormForm(BinOp(Conj, p, q)) =
    BinOp(Conj, conjNormForm(p), conjNormForm(q))
  | conjNormForm(BinOp(Disj, p, q)) =
    distribute(conjNormForm(p), conjNormForm(q));
```

Continuing our example:

```
- val cnf = conjNormForm(nnf);

val cnf =
  BinOp (Conj,BinOp (Conj,BinOp #,BinOp #),
              BinOp (Conj,BinOp #,BinOp #))
  : proposition

- print(display(cnf) ^ "\n");

(tv~pvtv~qv~tvr)^(~uv~pvtv~qv~tvr)^(tvqvtv~qv~tvr)^
(~uvqvtv~qv~tvr)^(tv~pvuv~qv~tvr)^(~uv~pvuv~qv~tvr)^
(tvqvuv~qv~tvr)^(~uvqvuv~qv~tvr)^(tv~pv~rv~qv~tvr)^
(~uv~pv~rv~qv~tvr)^(tvqv~rv~qv~tvr)^(~uvqv~rv~qv~tvr)
val it = () : unit
```

The output is getting hard to read, but it does not need to be human readable at this point—only computer readable.

Now for the actual tautology testing. At this point we assume that every proposition is either a disjunction of variables and negations of variables or the conjunction of such variables. If it is a top-level conjunction, then we check that every subproposition is a tautology. If it is a subproposition (disjunction), we check that it is a tautology by seeing if it has both a variable and its negation; we do that by collecting all the plain variables and all the variables that are negated and seeing if they have any overlap. To do all this, we write functions `positives` and `negatives` to collect those variables. We also rely on the functions `contains` from Section 2.7 and `intersection` from Exercise 2.7.6. We raise an exception if the proposition is not in conjunctive normal form.

```
exception NotInCNF of string;

fun positives(Var(s)) = [s]
  | positives(Neg(Var(s))) = []
  | positives(BinOp(Disj, p, q)) = positives(p) @ positives(q)
  | positives(x) = raise NotInCNF(display(x));

fun negatives(Var(s)) = []
  | negatives(Neg(Var(s))) = [s]
  | negatives(BinOp(Disj, p, q)) = negatives(p) @ negatives(q)
  | negatives(x) = raise NotInCNF(display(x));

fun tautCNF(BinOp(Conj, p, q)) = tautCNF(p) andalso tautCNF(q)
  | tautCNF(p) = intersection(positives(p), negatives(p)) <> [];
```

We could apply `tautCNF` directly to our value `cnf`, but to make future use easier, we write a function that will merely take a list of premises and a conclusion and will do all the other work for us.

```
- fun validArgument(premises, conclus) =
=     tautCNF(conjNormForm(negNormForm(cond(bigConj(premises),
=                                           conclus))));

val validArgument = fn : proposition list * proposition -> bool


- validArgument([cond(Var("p"), Var("t")),
=                cond(Neg(cond(Var("q"), Var ("t"))), Var("w")),
=                BinOp(Disj, Var("p"), Var("q")),
=                Neg(Var("w"))],
=               Var("t"));


val it = true : bool
```

This example and project were adapted from Paulson[22, pg 164–170].

Project

2.A The software in this section plays Game 2. It can be adapted to play Game 1 as well, if we make the observation that two propositions are logically equivalent ($p \equiv q$) if and only if their biconditional is a tautology ($p \leftrightarrow q \equiv T$). Write a helper function `bicond` which takes two propositions and creates a proposition in our system equivalent to $p \leftrightarrow q$. Then write a function `logEquiv` that takes two propositions and uses `bicond` and functions from this section to determine if the two propositions are logically equivalent.

2.B The "propositions" in this section of course are not truly propositions because they have variables in them. In order to evaluate a proposition (tell whether it is true or false), we need to have an *assignment* to the variables—that is, a specific true or false value assigned to each variable. Each row in a truth table represents an assignment to the variables. Consider this datatype:

```
datatype assignment = Assign of (string *
bool) list
```

Such an assignment associates variable names (`string`s) with truth values. Write a function `lookup` that takes an assignment and a string standing for a variable and returns a truth value assigned to that variable. Define an exception for it to raise if the string is not a bound variable. Then write a function `evaluateProposition` that takes a proposition and an assignment and determines its value.

2.C With `evaluateProposition`, it is possible to test if two propositions are logically equivalent by taking a "brute force" approach, similar to using a truth table. We would generate all possible assignments for a list of variables, evaluate both propositions for each assignment, and verify that the two propositions are always equal. Write a function `logEquiv2` that tests for logical equivalence in this way. Recommended helper functions are

- `enumerateAssignments`, which takes a list of variable names and produces a list of `assignments`, one for each possible combination of values. For example, `enumerateAssignments(["p", "q"])` would return `[Assign [("p",true),("q",true)], Assign [("p",false),("q",true)], Assign [("p",true),("q",false)], Assign [("p",false),("q",false)]]`.
- `testAllAssignments`, which takes two propositions and a list of assignments and evaluates each proposition for each assignment and verifies that the propositions are equal for each assignment.
- `collectVars`, which takes a proposition and produces a list of variable names. So that no variable occurs in the resulting list more than once, use the `union` operation from Exercise 2.7.6 or 2.7.8.

2.D A problem similar to the one in this section is that of determining whether a proposition is a *contradiction*—always false. The easy propositions for contradiction testing are those in *disjunctive normal form*—big disjunctions of subpropositions that each are conjunctions of variables or negations of variables. Write a set of functions that will turn a proposition into disjunctive normal form and then test whether the proposition is a contradiction.

Biography: Lawrence Paulson, 1955–

Lawrence Paulson is a computer scientist who has advanced the field of automatic theorem proving and logic. Having studied in the United States (California Institute of Technology and Stanford University), he has spent most of his career at Cambridge University. He has contributed to several software systems that produce proofs of theorems by automated reasoning. His research has included foundations of mathematics, such as set theory and symbolic logic. He also has developed a system for interactive verification of cryptographic protocols.