Unit Resolution Examples

Harry Bryant

May 2024

Contents

1 Inductive Definition of Unit Resolution

```
Inductive unitres : formula -> clause -> Prop :=
| subsumption : forall (c c2 : clause) (f : formula),
    In c f ->
    subset c c2 ->
    unitres f c2
| resolution : forall (c : clause) (l : literal)
    (f : formula),
    unitres f c ->
    is_literal_in_clause l c ->
    unitres f (cons (opposite l) []) ->
    unitres f (remove_literal_from_clause l c).
```

The unitres definition allows for a formula/clause pair to be defined using the Subsumption constructor. Once a Subsumption, it can be used as a resolution to remove a literal from a clause - where the negation of the literal exists as a unit clause in the formula. The functions used by the Sumbsumption and Resolution constructors ("In", "subset", and "is_literal_in_clause") produce a proposition and are defined as follows:

```
Definition memlc (l : literal) (c : clause) : Prop :=
  In l c.

Definition subset (c1 c2 : clause) : Prop :=
  forall l : literal, In l c1 -> In l c2.

Fixpoint is_literal_in_clause (l : literal)
    (c : clause) : Prop :=
  match c with
  | nil => False
  | hd :: tl => hd = l \/ is_literal_in_clause l tl
  end.
```

Note that "is_literal_in_clause" is a fixpoint rather than a regular definition because it is recursive.

2 Example 1: $\{\{a \vee b\}, \{\neg a\}\}$

```
Definition c1: clause := [pos "a"; pos "b"].
Definition c2: clause := [neg "a"].
Definition f: formula := [c1; c2].
```

In this example it is clear that we have a and $\neg a$ in the two clauses so they can be cancelled as both cannot be simultaneously true. As $\neg a$ is a unit clause (because the clause only contains one element) the unitres definitions can be applied.

2.1 Subsumption

We need to show that there is a Subsumption where the $c \in f$ and that $c \subset c'$. If these two checks pass then a unitres of f and c'. For these checks we will check that a clause is a subset of itself because in this case we are trying to subsume a clause that is in f and not a refined version.

```
Compute subsumption c1 c1 f.
Compute subsumption c2 c2 f.
```

The result of the first Subsumption is:

```
  = fun (x : In c1 f) (x0 : subset c1 c1) =>
    subsumption [pos "a"; pos "b"] [pos "a"; pos "b"]
      [[pos "a"; pos "b"]; [neg "a"]] x
       x0
  : In c1 f -> subset c1 c1 -> unitres f c1
```

- x is a proof that the clause $c1$ is present in f by asserting that $c1$ is an element of the list of clauses f.
- $x0$ is a proof $c1$ is a subset of itself.

However, this relies on the definitions of “In" and “subset" being correct which can be checked:

```
Compute In [pos "a"; pos "b"] [[pos "a"; pos "b"];
    [neg "a"]].
Compute In [pos "c"] [[pos "a"; pos "b"]; [neg "a"]].
```

```
  = [pos "a"; pos "b"] = [pos "a"; pos "b"]
     \/ [neg "a"]
      = [pos "a"; pos "b"] \/ False
  : Prop
```

The first check states that either $a \vee b = a \vee b$, or that $\neg a = a \vee b$, otherwise the statement is false and it is not a member. As the first case is met, true is returned, and $c1$ is in f.

```
    = [pos "a"; pos "b"] = [pos "c"] \/ [neg "a"]
       = [pos "c"] \/ False
    : Prop
```

The second check uses a clause c which obviously is not in f. The same check is made: either $a \vee b = c$, or that $\neg a = c$, otherwise the statement is false and it is not a member. As neither of the first two cases are met, c is not in f. This process can be repeated with all definitions used for example with subset:

```
   Compute subset c1 c1.

    = forall x : literal,
      pos "a" = x \/ pos "b" = x \/ False ->
       pos "a" = x \/ pos "b" = x \/ False
    : Prop
```

Here every literal in the first clause is going to either be an a or a b as these are the members of the second clause, which is trivially true. Therefore, because "In" and "subset" both hold (and have been shown to be correct) we can say that unitres f $c1$ holds - where unitres f $c1$ represents the formula/clause pair defined using the Subsumption constructor. The same can be done for the second Subsumption which results in unitres f $c2$.

The Inductive Definition can be used to prove that given the conditions being met, a Subsumption will be formed. From this Lemmas and Theorems can be proven to show that the definitions are correct. In order to actually compute Unit Resolutions decision procedures need to be defined which perform the same checks as the Inductive Resolution but produces a formula/clause pair.

We can use decision procedures to perform the Subsumption and add it to a list of Unit Resolutions that can be used future Resolutions:

```
Inductive formula_clause_pair : Set :=
| mk_fcp : formula -> clause -> formula_clause_pair.

Definition list_of_ures := list(formula_clause_pair).

Definition check_subsumption (c c2 : clause)
     (f : formula) : bool :=
  match memcf c f with
  | true => subset c c2
  | false => false
  end.

Definition append_to_list_of_ures
     (n : formula_clause_pair) (s : list_of_ures) :
         list_of_ures :=
  n :: s.
```

In order to have the decision procedures, a datatype to equivalent to the Inductive unitres must be defined. This is also Inductive but simply creates a pair from a formula and a clause to match the result produced by the original unitres definition - the difference here is we can compute it because it has been defined as a Set without the parameters which limit the original. Next is a definition to create a list of these pairs. The "check_subsumption" performs the checks of the Subsumption constructor in the original unitres Inductive Definition which returns true if "memcf" and "subset" both hold. There is also a definition to append a new element to the list.

```
Fixpoint literal_eq (l1 l2 : literal) : bool :=
  match l1, l2 with
  | pos s1, pos s2 => if string_dec s1 s2 then true
    else false
  | neg s1, neg s2 => if string_dec s1 s2 then true
    else false
  | _, _ => false
  end.

Fixpoint memlc (a:literal) (l:clause) : bool :=
    match l with
      | nil => false
      | b :: m => if literal_eq_dec a b then true
        else memlc a m
    end.

Fixpoint inb (l: literal) (c: clause) : bool :=
  match c with
  | [] => false
  | x :: xs => if literal_eq l x then true else inb l xs
  end.

(* Boolean version of subset *)
Definition subset (c1 c2: clause) : bool :=
  forallb (fun l => inb l c2) c1.
```

The decision procedures require definitions and functions that also produce a Boolean as opposed to a Proposition (See Section 1). The definition "memlc" represents the original "In". This uses "literal_eq". The "subset" definition uses "inb".

```
Definition compute_subsumption (c c2 : clause)
    (f : formula) (s : list_of_ures) : list_of_ures :=
  match check_subsumption c c2 f with
  | true => append_to_list_of_ures (mk_fcp f c2) s
  | false => s
  end.
```

The last definition is to add the unitres f $c1$ to a list of Unit Resolutions if the check definition passes. Testing it takes the same parameters as before but with the addition of an empty list of Unit Resolutions which will be appended to if the Subsumption is successful:

```
Compute compute_subsumption c1 c1 f s.

      = [mk_fcp [[pos "a"; pos "b"]; [neg "a"]]
          [pos "a"; pos "b"]]
      : list_of_ures
```

As the checks pass the unitres f $c1$ is added to the list of Unit Resolution (containing only this pair). The definition "mk_fcp" creates the pair so it functions as if it was a unitres. After doing the same for the second Subsumption we get:

```
      = [mk_fcp [[pos "a"; pos "b"]; [neg "a"]]
          [neg "a"];
         mk_fcp [[pos "a"; pos "b"]; [neg "a"]]
              [pos "a"; pos "b"]]
      : list_of_ures
```

2.2 Resolution

The two can now be resolved to cancel out a from the first clause in f and $\neg a$ from the second:

```
Compute resolution c1 (pos "a") f.

      = fun (x : unitres f c1)
          (x0 : is_literal_in_clause (pos "a") c1)
           (x1 : unitres f [opposite (pos "a")]) =>
        resolution [pos "a"; pos "b"] (pos "a")
          [[pos "a"; pos "b"]; [neg "a"]] x x0 x1
      : unitres f c1 ->
        is_literal_in_clause (pos "a") c1 ->
        unitres f [opposite (pos "a")] ->
        unitres f
         (remove_literal_from_clause (pos "a") c1)
```

Above is the result of the Resolution.

- x is a proof of unitres f $c1$, meaning it's evidence that $c1$ can be derived from f using Unit Resolution. Proven because we have shown $c1$ to be a Subsumption.

- $x0$ is a proof that a is a literal present in $c1$. It ensures that a is one of the literals being resolved in the Resolution step.

- $x1$ is a proof of unitres f $\neg a$, indicating that the clause [opposite (pos "a")] can be derived from f using Unit Resolution. This allows the cancellation of literals to take place. Because we have shown that unitres f $c2$ exists this holds as $c2 = \neg a$.

The function then applies the Resolution constructor to construct a proof of unitres f $(c1 \setminus a)$. Which in turn could be used for further Resolutions. Again it must be shown that the definitions are correct starting with "is_literal_clause":

```
Compute is_literal_in_clause (pos "a") c1.

     = pos "a" = pos "a" \/ pos "b" = pos "a" \/ False
     : Prop
```

Here, a must either equal $a \vee b$ otherwise the literal is not in the clause. The next definition to check is "removal_literal_from_clause":

```
Compute remove_literal_from_clause (pos "a")
     [pos "a"; pos "b"].

      = if lit_eq_dec (pos "a") (pos "a")
        then if lit_eq_dec (pos "a") (pos "b") then []
         else [pos "b"]
        else pos "a" :: (if lit_eq_dec (pos "a") (pos "b")
         then [] else [pos "b"])
     : clause
```

This check utilises the "lit_eq_dec" definition. If $a = a$ then a second check is made to see if $a = b$. if so then we are left with the empty list, otherwise we are left with b. This ensures that all instances of a are removed in one call. If the first check fails, then a is appended to the front of the result of the second check - if true then the result is a otherwise $a \vee b$.

Here the literal a was removed. The "is_literal_in_clause" definition applies "remove" which is apart of the Coq list library and is defined as follows:

```
Fixpoint remove (x : literal)
     (l : list literal){struct l} : list literal :=
       match l with
         | nil => nil
         | y::tl => if (lit_eq_dec x y) then
              remove x tl else y::(remove x tl)
       end.
```

The "remove" definition applies the "lit_eq_dec" definition (which will also be checked) to leave a list without any instances of x. It first checks to see if $a = a$. If so, then it removes a and checks if $a = b$. If so, the empty list is returned, otherwise the list only contains b. If in the first check $a \neq a$, then a is appended to the front of the result of checking if $a = b$ - which is the same as before. The next check is to ensure that "lit_eq_dec" functions correctly:

```
Compute lit_eq_dec (pos "a") (pos "a").
Compute lit_eq_dec (pos "a") (pos "b").
```

```
= lit_eq_dec (pos "a") (pos "a")
 : {pos "a" = pos "a"} + {pos "a" <> pos "a"}
```

The first computation returns that either $(a = a) \vee (a \neq a)$. Which obviously is satisfied by the left hand side - the result of which is true.

```
= lit_eq_dec (pos "a") (pos "b")
 : {pos "a" = pos "b"} + {pos "a" <> pos "b"}
```

The second returns that either $(a = b) \vee (a \neq b)$. Which satisfied by the right hand side so the result is false. Therefore, as these definitions are all shown to be correct, the overall Resolution is correct and resolving $c1 : a \vee b$ with $\neg a$ will result in $c1 : b$.

As with the Subsumption, we can compute the Resolution to remove the literal from the clause using the decision procedures that match the checks seen in the Inductive Definition:

```
Definition fcp_eq_dec
    (a b : formula_clause_pair) : bool :=
  match a, b with
  | mk_fcp f1 c1, mk_fcp f2 c2 =>
    if list_eq_dec
        (list_eq_dec literal_eq_dec) f1 f2 then
      if list_eq_dec literal_eq_dec c1 c2 then
        true
      else
        false
    else
      false
  end.

Fixpoint mempu (a:formula_clause_pair) (l:list_of_ures):
    bool :=
    match l with
      | nil => false
      | b :: m => if fcp_eq_dec a b then true
        else mempu a m
    end.

Definition check_resolution (c : clause) (l : literal)
    (f : formula) (s : list_of_ures) : bool :=
  match mempu (mk_fcp f c) s with
  | true =>
    match is_literal_in_clause l c with
    | true => match mempu (mk_fcp f [opposite l]) s with
      | true => true
      | false => false
      end
    | false => false
    end
  | false => false
  end.

Definition compute_resolution (c : clause) (l : literal)
    (f : formula) (s : list_of_ures) : clause :=
  match check_resolution c l f s with
  | true => remove_literal_from_clause l c
  | false => c
end.
```

The first definition needed for this is "fcp_eq_dec" which checks if two formula/clause pairs are equivalent. The next is "mempu" to check if the formula/clause pair is in the list of Unit Resolutions - replicating the check if the clause was already a Subsumption in the original unitres definition. The check resolution definition applies "mempu" and "is_literal_in_clause" and if these hold will return true. The "compute_resolution" definition removes the literal from the clause if true was returned.

```
Compute compute_resolution c1 (pos "a") f s2.

     = [pos "b"]
     : clause
```

Note that $s2$ is a list containing both Subsumptions.

```
Definition p1 : formula_clause_pair := mk_fcp f c1.

Compute mempu p1 s1.
Compute mempu p1 s.
```

The "mempu" fixpoint can be tested by creating an formula/clause pair of f and $c1$ to simulate a unitres. The first computation checks to see if the pair is in the list $s1$ which was defined to only contain $c1$. Therefore this is true. The second computation returns false because s is the empty list.

```
     = true
     : bool

     = false
     : bool
```

3 Example 2: $\{\{p \vee \neg q \vee r\}, \{q\}\}$

```
Definition c3: clause := [pos "p"; neg "q"; pos "r"].
Definition c4: clause := [pos "q"].
Definition f2: formula := [c3; c4].
```

3.1 Subsumption

We have to show that clauses $c3$ and $c4$ are Subsumptions. When computing $c3$ it returns:

```
= fun (x : In c3 f2) (x0 : subset c3 c3) =>
  subsumption [pos "p"; neg "q"; pos "r"]
    [pos "p"; neg "q"; pos "r"]
      [[pos "p"; neg "q"; pos "r"]; [pos "q"]] x x0
: In c3 f2 -> subset c3 c3 -> unitres f2 c3
```

- x is a proof that $c3$ is a member of $f2$. We have already shown in the previous example that the "In" function is correct (See section 2.1). As the definition of $f2$ includes $c3$ then this holds - $c3 \in f2$.
- $x0$ is a proof that $c3$ is a subset of $c3$ which is trivial.

```
= fun (x : In c4 f2) (x0 : subset c4 c4) =>
  subsumption [pos "q"] [pos "q"]
    [[pos "p"; neg "q"; pos "r"]; [pos "q"]] x x0
: In c4 f2 -> subset c4 c4 -> unitres f2 c4
```

The same can be said for the second Subsumption for the clause $c4$. It is defined in $f2$ and is a subset. Therefore, both clauses are defined as Unit Resolutions meaning they can be used in resolutions to derive further clauses.

Again, this can be computed using the Subsumption decision procedure (Section 2.1)

```
Compute compute_subsumption c3 c3 f2 s.

    = [mk_fcp [[pos "p"; neg "q"; pos "r"]; [pos "q"]]
        [pos "p"; neg "q"; pos "r"]]
    : list_of_ures
```

```
Compute compute_subsumption c4 c4 f2 s3.

    = [mk_fcp [[pos "p"; neg "q"; pos "r"];
        [pos "q"]] [pos "p"; neg "q"; pos "r"]]
    : list_of_ures
```

3.2 Resolution

Now that the two clauses are Subsumptions, the Resolution can take place to remove the literal:

```
Compute resolution c3 (neg "q") f2.

    = fun (x : unitres f2 c3)
        (x0 : is_literal_in_clause (neg "q") c3)
          (x1 : unitres f2 [opposite (neg "q")]) =>
        resolution [pos "p"; neg "q"; pos "r"] (neg "q")
          [[pos "p"; neg "q"; pos "r"]; [pos "q"]] x x0 x1
      : unitres f2 c3 ->
        is_literal_in_clause (neg "q") c3 ->
        unitres f2 [opposite (neg "q")] ->
        unitres f2
         (remove_literal_from_clause (neg "q") c3)
```

Above is the result computing the Resolution.

- x is a proof of unitres $f2$ $c3$, meaning it's evidence that $c3$ can be derived from $f3$ using Unit Resolution. Proven because we have shown it to be a Subsumption.
- $x0$ is a proof that $\neg q$ is a literal present in $c3$. In Section 2.2 we showed "is_literal_in_clause" to be correct. As we trust the definition and $\neg q$ can be seen in $c3$ this passes.
- $x1$ is a proof of unitres $f2$ q, indicating that the clause [opposite (neg "q")] can be derived from $f2$ using Unit Resolution. Because we have shown that $c4$ is a Subsumption, unitres $f2$ $c4$ exists.

These mean that the premises all hold and because we have shown that "remove_literal_from_clause" functions correctly the conclusion holds. Therefore, $\neg q$ is removed from $c3$ to leave $p \vee r$. Using the decision procedures (Section 2.2), and where $s4$ is a list containing both Subsumptions, is as follows:

```
Compute compute_resolution c3 (neg "q") f2 s4.

    = [pos "p"; pos "r"]
    : clause
```

The computation performs the checks which confirm that both $c3$ and $c4$ are both a unitres (as they are in the list of Unit Resolutions), and that $\neg q$ is in $c3$. Therefore it is removed and we are left with $p \vee q$.

4 Example 3: $\{\{p\} \wedge \{\neg p \vee q\} \wedge \{\neg q\}\}$

```
Definition c5: clause := [pos "p"].
Definition c6: clause := [neg "p"; pos "q"].
Definition c7: clause := [neg "q"].
Definition f3: formula := [c5; c6; c7].
```

4.1 Resolution Step 1

The first step is to show that the clauses used in the first Resolution step ($\{p\}, \{\neg p \vee q\}$) are Subsumptions to allow the removal to take place.

```
Compute subsumption c5 c5 f3.
Compute subsumption c6 c6 f3.

     = fun (x : In c5 f2) (x0 : subset c5 c5) =>
       subsumption [pos "p"] [pos "p"]
        [[pos "p"; neg "q"; pos "r"]; [pos "q"]] x x0
     : In c5 f2 -> subset c5 c5 -> unitres f2 c5

     = fun (x : In c6 f2) (x0 : subset c6 c6) =>
       subsumption [neg "p"; pos "q"] [neg "p"; pos "q"]
         [[pos "p"; neg "q"; pos "r"]; [pos "q"]] x x0
     : In c6 f2 -> subset c6 c6 -> unitres f2 c6
```

Both clauses $c5$ and $c6$ are confirmed as Subsumptions because they are both members of $f3$ when it was defined. They are then trivially subsets of themselves. Therefore, the first Resolution Step can take place:

```
Compute resolution c6 (neg "p") f3.

     = fun (x : unitres f3 c6)
        (x0 : is_literal_in_clause (neg "p") c6)
         (x1 : unitres f3 [opposite (neg "p")]) =>
       resolution [neg "p"; pos "q"] (neg "p")
        [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
         x x0 x1
     : unitres f3 c6 ->
       is_literal_in_clause (neg "p") c6 ->
       unitres f3 [opposite (neg "p")] ->
       unitres f3
        (remove_literal_from_clause (neg "p") c6)s
```

The Resolution takes $\neg p$ from $c6$. Both clauses involved have been defined as Subsumptions. The literal being removed is in the clause, therefore, a new clause ($\{q\}$) is produced. Once again, the decision procedures can be used to add the Subsumptions to the a list effectively containing elements of type unitres.

```
Compute compute_subsumption c5 c5 f3 s.
Compute compute_subsumption c6 c6 f3 s5.
```

The result of computing these two is a list with two formula-clause pairs:

```
    = [mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
        [neg "p"; pos "q"];
       mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
           [pos "p"]]
    : list_of_ures
```

Next is to compute the resolution to remove $\neg p$ from the clause $c6$:

```
Compute compute_resolution c6 (neg "p") f3 s6.
```

The result of this is just q as all the checks held.

```
    = [pos "q"]
    : clause
```

4.2 Resolution Step 2

Now the second Resolution step can take place to cancel out q and $\neg q$. Firstly, the clauses used in this Resolution need to be of type Unit Resolution:

```
Compute resolution c8 (pos "q") f3'.

    = fun (x : unitres f3' c8)
        (x0 : is_literal_in_clause (pos "q") c8)
         (x1 : unitres f3' [opposite (pos "q")]) =>
      resolution [pos "q"] (pos "q")
        [[pos "q"]; [pos "p"]; [neg "p"; pos "q"];
           [neg "q"]] x x0 x1
    : unitres f3' c8 ->
      is_literal_in_clause (pos "q") c8 ->
      unitres f3' [opposite (pos "q")] ->
      unitres f3'
       (remove_literal_from_clause (pos "q") c8)
```

- x is a proof of unitres $f3'$ $c8$, we have shown it to be a Subsumption so this passes.
- $x0$ is a proof that q is a literal present in $c8$.
- $x1$ is a proof of unitres $f3'$ $\neg q$, indicating that the clause [opposite (pos "q")] can be derived from $f3'$ using Unit Resolution. Because we have shown that $c7$ is a Subsumption and that q is a subset, therefore, unitres $f3'$ $c8$ exists.

Therefore, as the computation removes q from $c8$, the result is $\emptyset$ and the formula has been proven to be false. This can also be shown using the decision procedures. In order to do so though the clause produced in the first resolution must be added to the list of unit resolutions. An extra definition "is_resolution_complete" can be used:

```
Definition is_resolution_complete (c : clause)
    (l : literal) (f : formula) (s : list_of_ures) :
        list_of_ures :=
  let result := compute_resolution c l f s in
  match result with
  | [] => []
  | _  => append_to_list_of_ures (mk_fcp f result) s
  end.
```

This checks the result of the Resolution. If it is the empty clause then it returns an empty list indicating that falsity has been derived. Otherwise, the derived clause is added to the list of Subsumptions so that it can be used in further Resolutions.

```
Definition s8 := compute_subsumption c7 c7 f3 s7.
Compute s8.

     = [mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [pos "q"];
        mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [neg "p"; pos "q"];
        mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [pos "p"]]
     : list_of_ures
```

Because the result of the first resolution was not empty the derived clause is added to the list of Unit Resolutions ready for the next step - stored as $s7$.

```
Definition s8 := compute_subsumption c7 c7 f3 s7.
Compute s8.

     = [mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [neg "q"];
        mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [pos "q"];
        mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [neg "p"; pos "q"];
        mk_fcp [[pos "p"]; [neg "p"; pos "q"]; [neg "q"]]
            [pos "p"]]
     : list_of_ures
```

The first thing to do is check that the clause $c7$ is a Subsumption. Above shows that the list of Unit Resolutions now includes this extra clause. With

$s8$ containing the three previously subsumed clauses and the previously derived clause the Resolution can be computed.

```
Definition r4 := compute_resolution c8 (pos "q") f3 s8.
Compute r4.
```

This computation produces:

```
    = []
    : clause
```

The result of the "is_resolution_complete" defined is also the empty list, so in this case the proof is complete because we are left with false. Therefore, the proof is complete.