Introduction

Ce projet vise, en premier lieu, la transformation de formules de la logique propositionnelle en leurs formes normales disjonctives (FND) et conjonctives (FNC). Or, ces transformations « brutes » conduisant à des résultats non optimaux, il est nécessaire, en second lieu, d'implémenter des prédicats permettant de simplifier successivement ces formes.

Le langage de programmation imposé pour ce projet est SWI-Prolog. Une partie des fonctionnalités a déjà été implémentée et mise à disposition, notamment la définitions des connecteurs logiques : $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ ainsi que des prédicats assurant la transformation par équivalences d'une formule logique en sa FND.

Nous allons présenter dans ce rapport les différentes étapes de ce travail, à savoir :

1. La réalisation d'un prédicat permettant la transformation en FNC.
2. La conception de plusieurs prédicats simplifiant ces formes normales, afin d'obtenir des formes minimales.

1. Transformations de formules de $\mathcal{L}^0$ en FNC

Les transformations à effectuer sont les mêmes que pour la FND, sauf en ce qui concerne la distributivité. En effet, il faut utiliser la distributivité de la disjonction sur la conjonction, ce qui permet de générer une FNC, comme le montre l'algorithme suivant :

```
fonction distrDC (F : Formule) : Formule; {*}
debut
  si F se présente sous le format A & B V C, alors
    retourner distrDC(A V C) & distrDC(B V C)
  sinon si  F se présente sous le format A V B & C alors
    retourner distrDC(A V B) & distrDC(A V C)
  sinon si F se présente sous le format A & B alors
    retourner distrDC(A) & distrDC(B)
  sinon si F se présente sous le format A V B alors
    R1 ← distrDC(A);
    R2 ← distrDC(B);
    si R1 ou R2 est une conjonction de deux termes alors
       retourner distrDC(R1 V R2) { des distributions sont encore possibles }
    sinon
       retourner R1 V R2;
  fin si
  sinon retourner F;
fin;
```

* cette version permet d'éviter les résultats de type : a & (b V c) quand on veut, par exemple, transformer la formule a & b V c. Ceci est dû à l'associativité xfy de l'opérateur &. Cela n'a pas été prévu dans la version mise à disposition pour la distribution de la conjonction sur la disjonction, (voir le test ***testdstrc*** fourni dans nos jeux d'essai, cf. annexe A).

Voici l'implémentation en SWI-Prolog correspondant à cet algorithme :

```
% Distribution de la disjonction sur la conjonction
% permet de générer une FNC ( test : testDistrDC )
% distrDC(+Form, ?FormS)

distrDC((F1 & F2) 'V' F3, R1 & R2):- !,
      distrDC(F1 'V' F3, R1),
      distrDC(F2 'V' F3, R2).
distrDC(F1 'V' (F2 & F3), R1 & R2):- !,
      distrDC(F1 'V' F2, R1),
      distrDC(F1 'V' F3, R2).
distrDC(F1 & F2, R1 & R2):- !,
      distrDC(F1, R1),
      distrDC(F2, R2).
distrDC(F1 'V' F2, RS):-
      distrDC(F1, R1),
      distrDC(F2, R2),
      (functor(R1, &, _);
        functor(R2, &, _)), !,
      distrDC(R1 'V' R2, RS).
distrDC(F, F).
```

Et des traces d'exécution mettant en œuvre ce prédicat :

```
154 ?- distrDC(a 'V' b & c, F).

F = (a 'V' b) & (a 'V' c)

Yes
155 ?- distrDC(a 'V' b 'V' c & d, F).

F = (a 'V' b 'V' c) & (a 'V' b 'V' d)

Yes
156 ?- distrDC((a & b 'V' c) & (d 'V' (f 'V' g 'V' h & i) & j), F).

F = ((a 'V' c) & (b 'V' c)) & ((d 'V' f 'V' g 'V' h) & (d 'V' f 'V' g 'V' i)) & (d
'V' j)

Yes
```

Malheureusement, cela n'est pas encore suffisant puisque des problèmes liés à l'élimination des parenthèses superflues subsistent. Ces problèmes sont dus au traitement de l'associativité. Pour cette raison, nous proposons une nouvelle version du prédicat en question (voir le test *testassdg*, annexe A) :

```
% Elimination de toutes les parenthèses superflues dans une formule
% ne contenant que des conjonctions et disjonctions ( test : testAss )
% ass(+Form, ?FormS)

ass((F1 'V' F2) 'V' F3, F):- !,  % associativité gauche
      ass(F1, R1),               % les parenthèses liées à
      ass(F2, R2),               % l'associativité droite sont
      ass(F3, R3),               % éliminées automatiquement grâce à
      ass(R1 'V' R2 'V' R3, F). % l'associativité xfy de l'opérateur

ass(F1 'V' F2, R1 'V' R2):- !,    % traitement des sous termes
      ass(F1, R1),
      ass(F2, R2).
ass((F1 & F2) & F3, F):- !,       % associativité gauche
      ass(F1, R1),
      ass(F2, R2),
      ass(F3, R3),
      ass(R1 & R2 & R3, F).
ass(F1 & F2, R1 & R2):- !,        % traitement des sous termes
      ass(F1, R1),
      ass(F2, R2).
ass(F,F).
```

dont voici quelques traces d'exécutions :

```
158 ?- ass((((a & b) & c) 'V' d) 'V' f, F).

F = a & b & c 'V' d 'V' f

Yes
159 ?- ass((((a 'V' b) 'V' c) & (d & f)), F).

F = (a 'V' b 'V' c) & d & f

Yes
160 ?- ass(a & ((b 'V' c) 'V' (d 'V' (e & f))) 'V' (g), F).

F = a & (b 'V' c 'V' d 'V' e & f) 'V' g

Yes
```

A partir de là, il ne reste plus qu'à effectuer successivement les différents traitements pour transformer une formule en FNC:

```
fonction tFNC(F : Formule) : Formule;
debut
   F ← teq(F);
   F ← tim(F);
   F ← tng(F);
   F ← distrDC(F);
   F ← ass(F);
   retourner F;
fin;
```

```
% Transformation en une FNC sans simplification
% ( tests : testtFNC1, testtFNC2 )
% tFNC(+Form, ?Form)

tFNC(F,FNC):-
       teq(F, F1),
       tim(F1, F2),
       tng(F2, F3),
       distrDC(F3, F4),
       ass(F4, FNC).
```

Voici quelques traces d'exécutions du prédicat tFNC :

```
165 ?- tFNC('~'(('~'b '<->' '~'a) '->' c) 'V' d, F).

F = (b 'V' ~a 'V' d) & (a 'V' ~b 'V' d) & (~c 'V' d)

Yes
166 ?- tFNC('~'('~'b '<->' '~'a), F).

F = (~b 'V' ~a) & (~b 'V' b) & (a 'V' ~a) & (a 'V' b)

Yes
168 ?- tFNC(('~'('~'b '<->' '~'a) 'V' c) '<->' d, F).

F = (b 'V' ~a 'V' d) & (a 'V' ~b 'V' d) & (~c 'V' d) & (~d 'V' ~b 'V' ~a 'V' c) &
(~d 'V' ~b 'V' b 'V' c) & (~d 'V' a 'V' ~a 'V' c) & (~d 'V' a 'V' b 'V' c)

Yes
```

Le calcul des formes normales étant effectué, il ne reste plus qu'à les simplifier.

2. Simplification des formes normales

2.1 Elimination des répétitions de sous-termes

Les répétitions de sous-termes doivent être éliminées dans chaque terme d'une forme normale. Voici l'algorithme permettant de le faire pour une FND, celui pour une FNC étant analogue.

```
fonction elimineRepFND(F : FND) : FND;
debut
   si F se présente sous le format T V R alors
      T ← simpC(T);
      R ← elimineRepFND(R);
      retourner T V R;
   fin si
   sinon
      T ← simpC(T);
      retourner T;
    fin si
fin;

{ simpC permet de traiter un seul terme }

fonction simpC(T : TermeFND) : TermeFND;
debut
   si T se présente sous le format ST & R alors
      si estTermeC(ST, R) alors
         R ← simpC(R);
         retourner R
      fin si
      sinon
         R ← simpC(R);
         retourner ST & R
      fin si
   sinon
       T ← simpC(T) ;
      retourner T;
fin;

{ estTermeC permet de tester si un sous-terme appartient à un terme d'une FND }
```

```
% FND
% elimineRepFND(+FND, ?FNDS)

% Si au moins deux termes, on traite le premier terme et on
% recommence avec le reste
elimineRepFND(T 'V' R, TS 'V' RS):- !,
       simpC(T, TS),       % on traite le premier terme
       elimineRepFND(R, RS).     % on recommence avec le reste

% sinon on traite le terme
elimineRepFND(T, TS):-
       simpC(T, TS).

% terme d'une FND
% simpC(+Term, ?TermS)

% si au moins deux sous-termes, alors si le premier sous-terme existe
% dans le reste du terme, on l'élimine
% on recommence avec le reste
simpC(T & R, RS):-
       estTermeC(T, R), !,
       simpC(R, RS).
simpC(T & R, T & RS):- !,
       simpC(R, RS).
% sinon, on retourne le terme
simpC(T, T).
```

```
% tester l'appartenance d'un terme à une formule de conjonctions
% estTermeC(+Term, +Form)

estTermeC(T, T):- !.
estTermeC(T, T & _):- !.
estTermeC(T, _ & R):-
       estTermeC(T, R).
```

Voici un exemple d'exécution :

```
169 ?- elimineRepFND(a & d & b & b & a & c & d 'V' e & e 'V' f, F).

F = b & a & c & d 'V' e 'V' f

Yes
```

2.2 Elimination des termes faibles

On souhaite éliminer chaque terme dont il existe un terme plus fort ou équivalent dans la forme normale. Par exemple :

- pour la FND `a & b V a & c V a & b V b`, on doit obtenir `a & b V a & c`,
- pour la FNC `(a V b) & (a V c) & (a V b V c) & (a V b)`, on doit obtenir `(a V b) & (a V c)`.

Ceci permet aussi d'éliminer les répétitions de termes. Voici l'algorithme pour une FND :

```
fonction elimineFaiblesFND(F : FND) : FND;
debut
   si F se présente sous le format T V R alors
      si estFaibleFND(T, R) alors
         R ← eliminePlusFaiblesFND(T, R);
         R ← elimineFaiblesFND(R, R);
         retourner R;
      fin si
      sinon
         R ← elmiminePlusFaiblesFND(T, R);
         si R contient au moins un terme alors
            R ← elimineFaiblesFND(R),
            retourner T V R;
         fin si
         sinon
            retourner T;
      fin si
   sinon
   retourner F;
fin;

{ estFaibleFND permet de tester s'il existe un terme dans une FND qui est plus
  fort et non équivalent à un terme donné }

{  eliminePlusFaiblesFND  permet  d'éliminer  tous  les  termes  plus  faibles  ou
équivalents à un terme donné }
```

```
% FND
% elimineFaiblesFND(+FND, ?FNDS)

elimineFaiblesFND(T 'V' R, RS):-
       estFaibleFND(T, R), !,
       (eliminePlusFaiblesFND(T, R, RS1),
        elimineFaiblesFND(RS1, RS)
       );
        eliminePlusFaiblesFND(T, R, RS1), !,
        (elimineFaiblesFND(RS1, RS2), !,
         RS = T 'V' RS2
```

```
        );
         RS = T.
elimineFaiblesFND(T, T).


% FND
% estFaibleFND(+Term, +FND)

estFaibleFND(T1, T2 'V' R):-
       estFaibleC(T1, T2),
         not(estFaibleC(T2, T1)), !;
       estFaibleFND(T1, R).
estFaibleFND(T1, T2):-
       estFaibleC(T1, T2),
         not(estFaibleC(T2, T1)).


% terme avec conjonctions
% estFaibleC(+Term1, +Term2)

estFaibleC(F, T & R):-
       estTermeC(T, F), !,
       estFaibleC(F, R).
estFaibleC(F, T):-
       estTermeC(T, F).


% FND
% eliminePlusFaiblesFND(+Term, +FND, ?FNDS)

eliminePlusFaiblesFND(T1, T2 'V' R, RS):-
       estFaibleC(T2, T1), !,
       eliminePlusFaiblesFND(T1, R, RS).
eliminePlusFaiblesFND(T1, T2 'V' R, T2 'V' RS):-
       eliminePlusFaiblesFND(T1, R, RS), !.
eliminePlusFaiblesFND(_, T2 'V' _, T2).
eliminePlusFaiblesFND(T1, T2, T2):-
       not(estFaibleC(T2, T1)).
```

Voici un exemple d'exécution :

```
170 ?- elimineFaiblesFND(a & b & c 'V' a & b 'V' b & c 'V' d 'V' a & d 'V' a, F).

F = b & c 'V' d 'V' a

Yes
```

2.3 Elimination des termes contradictoires ou tautologiques

On doit éliminer les termes contradictoires dans les FNDs et les termes tautologiques dans les FNCs. L'algorithme pour les FNDs :

```
fonction elimineContrFND(F : FND) : FND;
debut
   si F se présente sous le format T V R alors
     si contr(T) alors
      retourner elimineContrFND(R)
     sinon
      retourner T V elimineContrFND(R)
   sinon
     si contr(F) alors
        retourner false
     sinon
        retourner F;
fin;
{ contr permet de tester si un terme d'une FND est contradictoire }

fonction contr(T : TermeDansFND) : Booléen;
debut
```

```
   si T contient un sous-terme et la négation de ce sous-terme alors
      retourner VRAI
   sinon
      retourner FAUX;
fin;
```

```
% Elimination des termes contradictoires dans une FND
% elimineContrFND(+FND, ?FNDS)


% cas où la FND contient au moins 2 termes :

% si le premier terme d'une FND est contradictoire alors on l'élimine
% et on recommence avec le reste de la formule
elimineContrFND(T 'V' R, RS):-
      contr(T), !,
      elimineContrFND(R, RS).

% sinon on garde le premier terme et on recommence avec le reste,
% si le traitement de celui-ci conduit à 'false' alors on l'élimine
% (pour éviter d'avoir des résultats de type : Term1 V .. V false)

elimineContrFND(T 'V' R, T 'V' RS):-
      elimineContrFND(R, RS),
      RS \== false, !.
elimineContrFND(T 'V' _, T).     % on élimine false


% cas où la FND contient un seul terme :

% le résultat est 'false' si le terme est contradictoire
elimineContrFND(T, false):-
      contr(T), !.

% on retourne le terme sinon
elimineContrFND(T, T).


% tester si un terme d'une FND est contradictoire
% contr(+Term)

contr('~'T & R):-
      estTermeC(T, R), !.
contr(T & R):-
      estTermeC('~'T, R), !.
contr(_ & R):-
      contr(R).
```

Voici deux exemples d'exécutions :

```
173 ?- elimineContrFND(a & '~'b & b 'V' c & a 'V' a & b & c & d & '~'b 'V' a & '~'
a, F).

F = c & a

Yes
174 ?- elimineContrFND(a & b & '~'a 'V' b & '~'b, F).

F = false

Yes
```

2.4 Tri alphabétique des sous-termes

Un des objectifs du projet est d'implémenter un prédicat pour effectuer un classement alphabétique des sous-termes. Il faut en particulier gérer la négation de sorte qu'on ait :

a > ¬a > b > ¬b > c ...

Voici l'implémentation de ce prédicat en SWI-Prolog :

```
% FND
% alphaFND(+FND, ?FNDS)

% Si au moins deux termes, on traite le premier terme et on
% recommence avec le reste
alphaFND(T 'V' R, TS 'V' RS):- !,
      alphaC(T, TS),      % on traite le premier terme
      alphaFND(R, RS).    % on recommence avec le reste

% sinon on traite le terme
alphaFND(T, TS):-
      alphaC(T, TS).



% ordonner alphabétiquement une formule avec conjonctions
% alphaC(+Form, ?FormS)

% si au moins deux termes, alors on déplace le terme le plus petit
% (alphabétiquement) au début et on recommence avec le reste
alphaC(T & R, M & RS):- !,          % au moins deux termes
      extraitMinC(T & R, M, R1),    % on extrait le minimum et le
                                    % reste de la formule
      alphaC(R1, RS).               % on recommence avec le reste
                                    % calculé

% sinon, on retourne le terme
alphaC(T, T).



% trouver le terme le plus petit dans une formule avec conjonctions
% et renvoyer la formule sans ce terme

% Pré-condition : au moins deux termes, car on ne peut pas renvoyer
% une formule vide en cas d'un seul élément
extraitMinC(T & T2 & R, M, T & RS):-
      extraitMinC(T2 & R, M, RS),
      inf(M, T), !.
extraitMinC(T & T2 & R, T, T2 & R):-!.
extraitMinC(T & T2, T, T2):-
      inf(T, T2), !.
extraitMinC(T & T2, T2, T).


% tester si un terme est alphabétiquement inférieur ou égal à un autre
% + gestion particulière pour les termes commençant par une négation
% inf(+Term1, +Term2)

inf('~'A, '~'B):- !,
      A @=< B.
inf(A, '~'B):- !,
      A @=< B.
inf('~'A, B):- !,
      A @< B.
inf(A, B):-
      A @=< B.
```

```
178 ?- a 'V' '~'c & c & d, F).  & c & p & '~'b & c & y 'V' '~'d & a & e & b & '~'a
'V' '~'c & c & d, F).

F = ~b & c & c & d & h & p & y 'V' a & ~a & b & ~d & e 'V' c & ~c & d

Yes
```

2.5 Synthèse

Nous pouvons maintenant appliquer ces simplifications à des formes normales pour les optimiser. Voici un algorithme qui applique successivement toutes ces simplifications sur une FND :

```
fonction simpFND(F : FND) : FND;
debut
   F ← elimineRepFND(F);
   F ← elimineFaiblesFND(F);
   F ← elimineContrFND(F);
   si non(taut(F)) alors
      F ← alphaFND(F); { ordonner les sous-termes }
      F ← alphaD(F); { ordonner partiellement les termes }
      retourner F
   fin si
   sinon
      retourner true;
fin;
```

```
% Simplification d'une FND ( test : testSimpFND1, testSimpFND2 )
% simpFND(+FND, ?FNDS)

simpFND(F, FS):-
      elimineRepFND(F, FS1),          % on élimine les répétitions dans des
                                      % sous-termes
      elimineFaiblesFND(FS1, FS2),    % on élimine les termes faibles
      elimineContrFND(FS2, FS3),      % on élimine les termes contradictoires
      not(taut(FS3)), !,              % le résultat ne doit pas être 'true'
      alphaFND(FS3, FS4),             % on ordonne les sous-termes
      alphaD(FS4, FS).                % on ordonne les termes
simpFND(_, true).
```

```
180 ?- simpFND(d & '~'c & b & d 'V' '~'b & '~'a 'V' '~'a & d & '~'b 'V'c & e & a &
'~' c, F).

F = b & ~c & d 'V' ~a & ~b

Yes
181 ?- simpFND(a & '~'a 'V' a & b & c & '~'b, F).

F = false

Yes
182 ?- simpFND(a & b & '~'b 'V' c 'V' c & f & g 'V' '~' c 'V' d, F).

F = true

Yes
```

Conclusion

Différentes améliorations semblent encore possibles. Plusieurs prédicats ayant les mêmes fonctionnalités ont été élaborés pour chacun des deux types de forme normale (FND/FNC). On pourrait factoriser cela en manipulant des listes de termes au lieu de formules et en implémentant des prédicats pour convertir ces listes de termes en formules et réciproquement. Ceci permettrait, en outre, d'améliorer les performances.

Pour chaque simplification élémentaire, nous avons implémenté un prédicat qui parcourt la totalité de la formule passée en argument. On pourrait réduire le temps d'exécution en faisant plusieurs simplifications en un seul parcours, et donc avec un seul prédicat. Par exemple : un seul prédicat pour éliminer les répétitions des sous-termes et les mettre en ordre alphabétique. Nous n'avons cependant pas opté pour cette solution. En effet, nous souhaitions pouvoir effectuer chaque simplification indépendamment des autres, et ainsi garder la possibilité de choisir différentes combinaisons de ces simplifications élémentaires.

ANNEXES

A- Liste des tests disponibles

testdistrCD/0	distributivité du $\wedge$ sur le $\vee$
testass/0	associativité
testtFND1/0	transformation en FND
testtFND2/0	transformation détaillée en FND
testtFNC1/0	transformation en FNC
testtFNC2/0	transformation détaillée en FNC
testsimpFND1/0	simplification d'une FND
testsimpFND2/0	simplification détaillée d'une FND
testsimpFNC1/0	simplification d'une FNC
testsimpFNC2/0	simplification détaillée d'une FNC
testsFND/0	transfo. et simplif. détaillées en FND
testsFNC/0	transfo. et simplif. détaillées en FNC
test/0	test général avec entrées en ligne
testdstrc/0	comparaison dstrc et distrCD
testassdg/0	comparaison assd, assg et ass

B- Présentation hiérarchique des prédicats

```
sFND(+Form, ?FNDS)
sFNC(+Form, ?FNCS)

        tFND(+Form, ?FND)
        tFNC(+Form, ?FNC)
              teq(+Form, ?FormS)
              tim(+Form, ?FormS)
              tng(+Form, ?FormS)
              distrCD(+Form, ?FormS)
              distrDC(+Form, ?FormS)
              ass(+Form, ?FormS)

        simpFND(+FND, ?FNDS)
        simpFNC(+FNC, ?FNCS)

              elimineRepFND(+FND, ?FNDS)
              elimineRepFNC(+FNC, ?FNCS)
                    simpC(+Term, ?TermS)
                    simpD(+Term, ?TermS)

              elimineFaiblesFND(+FND, ?FNDS)
              elimineFaiblesFNC(+FNC, ?FNCS)
                    estFaibleFND(+Term, +FND)
                    estFaibleFNC(+Term, +FNC)
                    estFaibleC(+Term1, +Term2)
                    estFaibleD(+Term1, +Term2)
                    estTermeC(+Term, +Form)
                    estTermeD(+Term, +Form)
                    eliminePlusFaiblesFND(+Term, +FND, ?FNDS)
                    eliminePlusFaiblesFNC(+Term, +FNC, ?FNCS)

              elimineContrFND(+FND, ?FNDS)
              elimineTautFNC(+FNC, ?FNCS)
                    contr(+Term)
                    taut(+Term)

              alphaFND(+FND, ?FNDS)
              alphaFNC(+FNC, ?FNCS)
                    alphaC(+Form, ?FormS)
                    alphaD(+Form, ?FormS)
                    extraireMinC(+Form, ?TermS, ?FormS)
                    extraireMinD(+Form, ?TermS, ?FormS)
                    inf(+Term1, +Term2)
```

C- Listing du programme