

Simplifying Logical Formulas

Assume we have a SCHEME logical expression involving the operators "and", "or", "not", and the truth constants #t and ().

Example: (and a (not b) (or b c) (not (and c (not d)))) #t (and e) (or () c e))

Such expressions can be simplified in a variety of ways.

Negations: Use DeMorgan's Laws and the double negation rule to drive negations "inward" to apply only to atoms – unsimplified Negation Normal Form – NNF

$$(\text{not } (\text{and } e_1 e_2 \dots e_n)) = (\text{or } (\text{not } e_1) (\text{not } e_2) \dots (\text{not } e_n))$$

$$(\text{not } (\text{or } e_1 e_2 \dots e_n)) = (\text{and } (\text{not } e_1) (\text{not } e_2) \dots (\text{not } e_n))$$

$$(\text{not } (\text{not } e)) = e$$

Assume that we have a function called demorgan that does these simplifications.

In the example above,

$$(\text{demorgan } '(\text{and } a (\text{not } b) (\text{or } b c) (\text{not } (\text{and } c (\text{not } d))) \#t (\text{and } e) (\text{or } () c e))) = \\ (\text{and } a (\text{not } b) (\text{or } b c) (\text{or } (\text{not } c) d) \#t (\text{and } e) (\text{or } () c e))$$

Further Simplifications

$$[\text{nnf1}] \quad (\text{or } () < args >) = (\text{or } < args >) \quad (\text{or } \#t < args >) = \#t$$

$$[\text{nnf1}] \quad (\text{and } \#t < args >) = (\text{and } < args >) \quad (\text{and } () < args >) = ()$$

$$[\text{nnf0}] \quad (\text{or}) = () \quad (\text{and}) = \#t$$

$$[\text{nnf4}] \quad (\text{or } X (\text{or } Y Z)) = (\text{or } X Y Z)$$

$$[\text{nnf4}] \quad (\text{and } X (\text{and } Y Z)) = (\text{and } X Y Z)$$

$$[\text{nnf2}] \quad (\text{or } X \dots < args > \dots X) = (\text{or } < args > \dots X)$$

$$[\text{nnf2}] \quad (\text{and } X \dots < args > \dots X) = (\text{and } < args > \dots X)$$

$$[\text{nnf3}] \quad (\text{or } X) = X \quad (\text{and } X) = X$$

- Each simplification applies only to a conjunction or disjunction.
- We do not worry about the arguments.
- We do not worry about any containing expression
of which the conjunction or disjunction may be an argument.

Now let's program nnf0, nnf1, nnf2, nnf3, and nnf4.

```
(define nnf0 (lambda (e) ; reduce (and) and (or) to #t and #f
  (cond ((equal? e '(and)) #t)
        ((equal? e '(or)) #f)
        (#t e) ) ))
```

```
(define nnf2 (lambda (e) ; eliminate duplicate atomic arguments
  (cond ((not (pair? e)) e)
        ((eq? (car e) 'not) e)
        (#t (cons (car e) (makeset (cdr e)))) ) ))
```

```
(define makeset (lambda (l)
  (cond ((null? l) ())
        ((null? (cdr l)) l)
        (#t (union (list (car l)) (makeset (cdr l)))) ) ))
```



```

(define nnf3 (lambda (e)          ; (and x) = (or x) = x
  (cond ((not (pair? e)) e)
        ((and (or (eq? (car e) 'and) (eq? (car e) 'or))
              (null? (cddr e)))
         (cadr e) )
        (#t e) ) ) )

```

```

(define nnf4 (lambda (e)          ; (or a b (or c d)) = (or a b c d)
  (cond ((not (pair? e)) e)      ; (and a b (and c d)) = (and a b c d)
        ((eq? (car e) 'not) e)
        (#t (cons (car e) (purge (car e) (cdr e)))) ) ) )

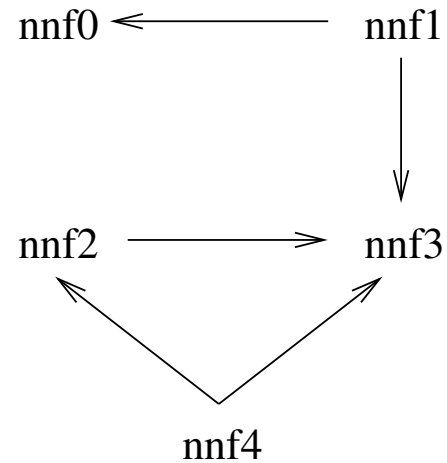
```

```

(define purge (lambda (op args)  ; does the real work of nnf4
  (cond ((null? args) ())
        ((not (pair? (car args))) (cons (car args) (purge op (cdr args))))
        ((eq? op (caar args)) (append (cdar args) (purge op (cdr args))))
        (#t (cons (car args) (purge op (cdr args)))) ) ) )

```

In what order should nnf0, nnf1, nnf2, nnf3, and nnf4, be applied?

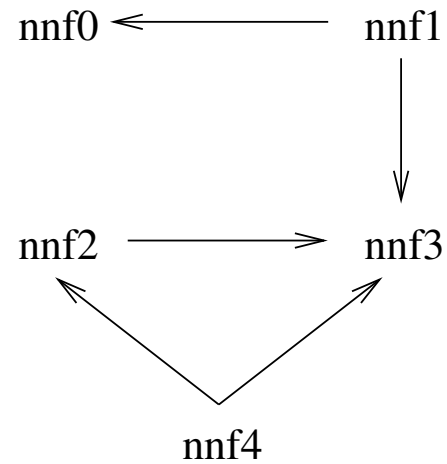


nnf0 ← nnf1:

Eliminating constant arguments of #t and () can transform the entire expression into (and) or (or). Notice that nnf0 does NOT create expressions applicable to nnf1. nnf0 reduces (and) to #t, which is not a disjunction or a conjunction. (It might be the argument of such a containing expression.)

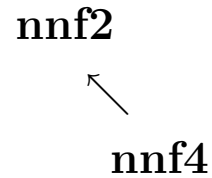
nnf1
↓
nnf3

Eliminating #t and () can create and's and or's with single arguments.

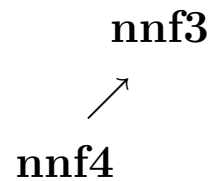


nnf2 \longrightarrow nnf3

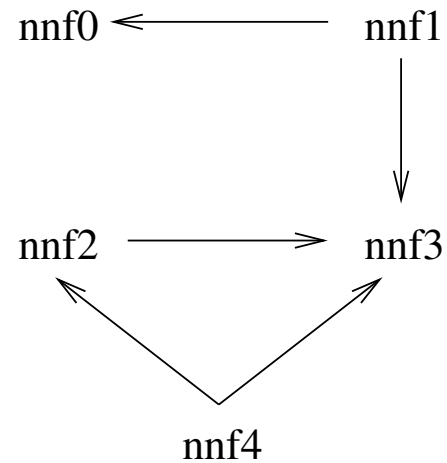
Eliminating duplicate arguments can create and's and or's with single arguments.



Combining or's of or's (and's of and's) can create duplicate arguments.



Combining or's of or's (and's of and's) can create single argument and's/or's.



If $\text{nnf}_i \longrightarrow \text{nnf}_j$, then nnf_i must be applied prior to nnf_j .

The above graph represents a partial order. This means that there is a linear order of the functions such that: Whenever there is a path from nnf_i to nnf_j in the graph, then nnf_i precedes nnf_j in the linear order, (and NOT vice versa).

One such linear order is: $\text{nnf}_1, \text{nnf}_0, \text{nnf}_4, \text{nnf}_2, \text{nnf}_3$

This tells us how to simplify a near-NNF formula using these functions.

```
(define nnf (lambda (e) (nnfaux (demorgan e))))
```

Remember, we assume the function `demorgan` to push negations inward and remove double negations.

So `nnfaux` will now do the remaining simplification work.

Note that we cannot merely call `nnf0` – `nnf4` on the expression. The dependency graph only describes what happens with an expression and its immediate arguments. So we have to make sure that the arguments themselves are fully simplified first. Then we simplify the expression at hand.

```
(define nnfaux (lambda (e)
  (cond ((not (pair? e)) e)
        (#t (let ((simpargs (map nnfaux e)))
              (nnf3 (nnf2 (nnf4 (nnf0 (nnf1 simpargs)))))))))
```