v.7.8

XOR Sat Solving

<main> ::=

> <imports> <sat-solve> <provide>

<imports> ::=

```
(require "multinomial-reduction-utils.rkt")
(require (only-in "boolean-multinomial-reduction.rkt" add mult))
(require racket/match)
```

The boolean satifiability problem can be phrased as checking whether a boolean-valued function on n-variables is in fact the constant 0 function. However, the language that we use when solving SAT problems is incongruous with this interpretation, since a given boolean function may be expressible in multiple ways in terms of logical and ($\wedge$) and logical or ($\vee$). We see this in the existence of the conjunctive normal form and disjunctive normal forms. But if we were to instead use the algebra of logical and and xor, then we would have a one to one relation between boolean functions and expressions. This leads to a simple and elegant algorithm to check whether a quantifier free formula on boolean variables is satisfiable.

To get an idea of why this may be helpful consider the following workflow. Start with a formula with operations $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\oplus$(xor), and $\Leftrightarrow$, such as $(\Rightarrow (x_0 \wedge x_1)(x_0 \wedge x_1))$, where x_0 and x_1 are boolean variables. Transform the formula into one with only $\wedge$ and $\oplus$. Then simplify the resultant multinomial purely via equational laws of the xor algebra. If the final formula is anything but 0, the constant 0 function, then the original formula is satisfiable.

To write the equational laws, I will write $\oplus$ as $+$ and $\wedge$ as *, and I

will sometimes leave multiplication implicit. We have commutativity and associativity – $A + B = B + A$, $AB = BA$, $(A + B) + C = A + (B + C)$, and $(AB)C = A(BC)$. We have top and bottom elements 0 and 1, $A * 0 = 0$, $A * 1 = A$, and $AA = A$. Multiplication distributes over addition – $A(B + C) = AB + AC$ and $(A + B)C = AC + BC$. But we had all of this in the $\wedge / \vee$ calculus. What is new is additive identity since $A + 0 = A$ and additive inverses since $A + A = 0$. In other words, we are working over a field with two elements.

What does this buy us? Well first of all it justifies our use of $\oplus$ as $+$ and $\wedge$ as *. But more importantly it immediately points toward an algorithm for SAT solving. Namely, convert your propositional logic to $\oplus / \wedge$ form, and multiply. If the final result is 0, your formula is UNSAT. Anything else is satisfiable, and a result of 1 entails validity.

To transform the formula into xor form, we can derive the following equivalences.

$$\neg A \Leftrightarrow 1 + A$$

$$A \vee B \Leftrightarrow A + B + AB$$

$$(A \Rightarrow B) \Leftrightarrow \neg A \vee B \Leftrightarrow \neg A + B + (\neg A)B$$

$$\Leftrightarrow (1 + A) + B + (1 + A)B$$

$$\Leftrightarrow 1 + A + B + B + AB \Leftrightarrow 1 + A + AB$$

$$(A \Leftrightarrow B) \Leftrightarrow \neg(A + B) \Leftrightarrow 1 + A + B$$

<sat-solve> **::=**

```
(define (compile-to-xor formula)
  (if (symbol? formula) (make-mset (symbol->string formula))
      (let ([mset-one (make-mset '((0 . 1)))])
        (if (eq? 2 (length formula))
            (add mset-one (compile-to-xor (cadr formula)))
            (let ([x (compile-to-xor (cadr formula))]
```

```
                [y (compile-to-xor (caddr formula))])
            (match (car formula)
              ['¬   (add mset-one x)]
              ['∧   (mult x y)]
              ['∨   (add x y (mult x y))]
              ['⇒   (add mset-one x (mult x y))]
              ['⊕   (add x y)]
              ['⇔   (add mset-one x y)])))))))
```

Recall the example $x_0 \wedge x_1 \Rightarrow x_0 \wedge x_1$. After translating to xor by recursively applying the rule for implication above, we get $1 + x_0 * x_1 + (x_0 * x_1) * (x_0 * x_1)$. Then:

$1 + x_0 * x_1 + (x_0 * x_1) * (x_0 * x_1)$

$1 + x_0 * x_1 + x_0 * x_1$ by $AA = A$

$1 + 0$ by $A + A = 0$

0 by $A + 0 = A$

Here is the output of our corresponding compile-to-xor function:

Examples:

```
> (require "sat-solver.rkt" "multinomial-reduction-utils.rkt")
> (compile-to-xor '(⇒ (∧ v0 v1) (∧ v0 v1)))
(multiset (↥ (multiset)))
> (view-multinomial (compile-to-xor '(⇒ (∧ v0 v1) (∧ v0 v1))))
"1"
```

Note that compile-to-xor does not output 1, but rather a nested multiset. We use an internal representation of multinomials that more directly reflects their equivalences due to their equational laws. To understand this, please read "multinomial-reduction.rkt", or run "scribble –html multinomial-reduction.rkt" and read the html output.

We have not only shown that the original formula is satisfiable, but also that it is the constant 1 function, or valid! (Check out "bool-test.rkt" for more examples.) What's more we have done it with

very little code in a conceptually clear way. Next I hope to explore the relationship between SAT and the derivatives of their multinomial representations.

<provide> **::=**

```
(provide compile-to-xor)
```