Reduction to SAT: A First Experience

By Calvin Greve

calvin.greve@corratio.com

Part 1: Background information

- Background
- The expression language
- Boolean Satisfiability
- Verification Approaches

I saw my dad’s screen
He was programming in lisp
Just parenthesis

Part 2: The project

- The Four Passes
- Commutative diagram
- Proof Approaches
- Sets
- Rules Classes
- Tools
- Conclusion

Background

- Senior at Coe College, pursuing a Computer Science degree
- For my practicum I did an internship at Corratio.
- The challenge of this internship was to develop and verify software that transforms a simple expression language into SAT.

CORRATIO
TECHNOLOGIES

Expression Language with binding:

- I defined an expression language using a recursive structure. Each expression can contain combinations of either sub-expressions, or variables.
- For the sake of simplicity, expressions were limited to 2 arguments where applicable.
- Let bindings were the source of the majority of the issues I encountered during this project

Expressions:

- Var
  - Symbols or Integers
  - Boolean Variables (T/F)
- (NOT expr)
- (AND expr1 expr2)
- (OR expr1 expr2)
- (XOR expr1 expr2)
- (NAND expr1 expr2)
- (ITE expr1 expr2 expr3)
  - If-Then-Else
- (LET var arg body)
  - Var is a variable
  - Arg and Body are both expressions
  - Binds variable to the argument within the body

Boolean Satisfiability Problems

The Boolean Satisfiability problem comes down to the question:

> Is there some truth assignment to the inputs of an expression that makes it true.

Expressible in Conjunctive Normal Form (CNF)

K-SAT is a subset of boolean satisfiability problems wherein the number of literals is no greater than K.

Every boolean satisfiability problem can be reduced to 3-SAT. However this may not maintain equivalence with the original expression.

Satisfiable: (!X + !Y)

X is False || Y is False

Unsatisfiable: (X) & (Y) & (!X + !Y)

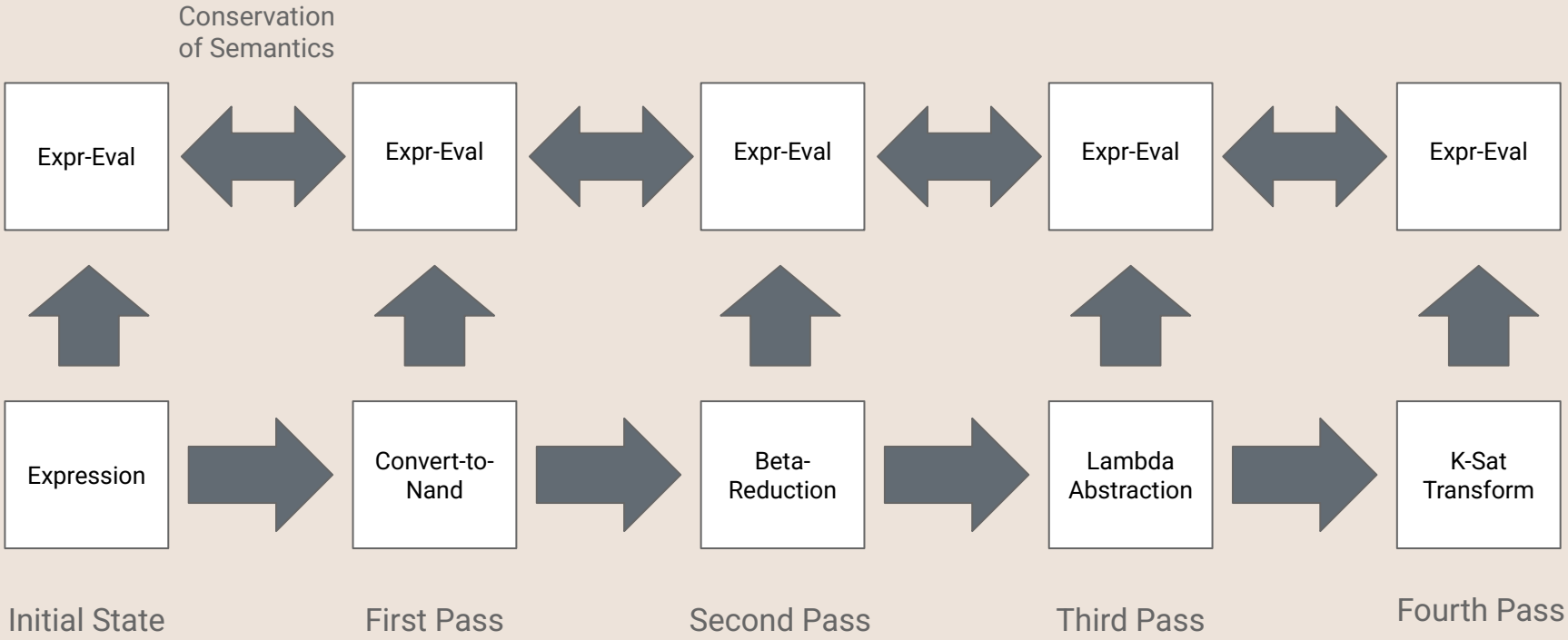
Verification Approaches:

How Close does it get us?	Verification Approach:	Downsides
• Proves correctness for input expressions	Testing	• Execution time increases exponentially with number of variables. • Needs to be re-run for every test • 2^n different variable combinations
• Proves correctness for all possible variable combinations. • Proves correctness for input expressions	BDD	• Execution time and size increases exponentially with number of variables. • Needs to be re-run for every test.
• Proves correctness of the transformation algorithm itself. • Proves correctness for all possible input expressions • Only needs to be executed once	Proof	• Requires substantial human effort.

The Four Passes

The 4 Passes	Necessary Proofs
Pass 1: Convert to Nand	Semantics + Structure, Induction scheme
Pass 2: Spinal Alignment	Beta-Reduction, Alpha-Conversion, Sets, Genvar, Semantics + Structure, Custom Induction scheme
Pass 3: Lambda Abstraction	“Evisceration”, “Reconstruction”, Lambda-Bindings, Semantics + Structure
Pass 4: Reduction to SAT	Semantics + Structure

Commutative Diagram



Proof Approaches

- Induction
  - Simple
  - Merged
  - Custom
- Congruences
  - Some proofs resisted induction
  - Congruence was used to simplify the environment

```
(defthm beta-reduce-let-conserves-semantics
    (implies
     (and (varp var)
          (weak-nand-exprp arg)
          (weak-nand-exprp body)
          (disjoint (bound-var-finder body)
                    (var-finder arg)))
     (equal (expr-eval (beta-reduce-let var arg body) env)
            (expr-eval body (acons var (expr-eval arg env) env))))
  :hints (("goal" :induct (list (beta-reduce-let-induction var arg body env))
                  :do-not-induct t
                  :do-not '(generalize fertilize eliminate-destructors))
          (pattern::hint
           (:and
            (:literally body)
            (:term (expr-eval body e)))
           :expand ((expr-eval body e)))))
```

The Environment:

```
(def-universal-equiv assoc-equiv!
    :qvars (key)
    :equiv-terms ((equal (assoc key x))))

(defthm assoc-equiv-implies
    (implies
     (assoc-equiv! x y)
     (equal (assoc key x)
            (assoc key y)))
  :rule-classes (:congruence)
  :hints (("goal" :use assoc-equiv!-necc)))
```

Equivalence of values

```
(defthm different-commutes
    (implies
     (not (equal a b))
     (assoc-equiv! (cons (cons a x) (cons (cons b y) env))
                   (cons (cons b y) (cons (cons a x) env))))
  :rule-classes ((:rewrite :loop-stopper ((a b))))
  :hints (("Goal" :in-theory (e/d (assoc-equiv!) nil))))

(defthm overwrite-clobbers
    (assoc-equiv! (cons (cons a x) (cons (cons a y) env))
                  (cons (cons a x) env))
  :hints (("Goal" :in-theory (e/d (assoc-equiv!) nil))))
```

Commutation & Overwrite

```
(defthm rewrite-assoc-equiv
    (implies
     (and (equal a b)
          (equal v1 v2)
          (assoc-equiv! env1 env2))
     (equal (assoc-equiv! (cons (cons a v1) env1)
                          (cons (cons b v2) env2))
            t))
  :hints ((pattern::hint
           (not (assoc-equiv! x y))
           :in-theory (enable assoc-equiv!)
           :restrict ((assoc-equiv! ((x x) (y y)))))))
```

Equivalence of Environments

Proof Approaches

- Induction
  - Simple
  - Merged
  - Custom
- Congruences
  - Some proofs resisted induction
  - Congruence was used to simplify the environment

```
(defthm beta-reduce-let-conserves-semantics
    (implies
     (and (varp var)
          (weak-nand-exprp arg)
          (weak-nand-exprp body)
          (disjoint (bound-var-finder body)
                    (var-finder arg)))
     (equal (expr-eval (beta-reduce-let var arg body) env)
            (expr-eval body (acons var (expr-eval arg env) env))))
  :hints (("goal" :induct (list (beta-reduce-let-induction var arg body env))
                  :do-not-induct t
                  :do-not '(generalize fertilize eliminate-destructors))
          (pattern::hint
           (:and
            (:literally body)
            (:term (expr-eval body e)))
           :expand ((expr-eval body e)))))
```

Sets

- My set library primarily utilized quantification over membership or member count
  - Subset
  - Disjoint
  - No-Duplicates
  - Sub-Bag
- Most proofs revolved around list operations and how they worked under quantification
  - Member
  - Cons
  - Append
  - Remove

```
(def-universal-equiv set-equiv!
    :qvars (item)
    :equiv-terms ((iff (member item x))))
```

```
(defthm subset-implies
    (implies (and (subset x y)
                  (member a x))
             (member a y))
  :hints (("goal" :use subset-necc))
  :rule-classes (:rewrite
                 :forward-chaining))
```

```
(defthm disjoint-implies
    (implies
     (and (disjoint x y)
          (member a x))
     (not (member a y)))
  :hints (("goal" :use disjoint-necc))
  :rule-classes (:rewrite :forward-chaining))
```

```
(defthm no-duplicates-implies
    (implies (no-duplicates l)
             (<= (atom-count x l) 1))
  :hints (("goal" :use no-duplicates-necc))
  :rule-classes (:rewrite :linear))
```

The Rule Classes

I primarily utilized 4 rule classes:

1. Rewrite
2. Congruence
3. Forward-Chaining
4. Linear

- Rewrite
  - Allows the theorem prover to rewrite statements that are an exact syntactic pattern match.

```
(defthm convert-to-nand-preserves-semantics
    (equal (expr-eval (convert-to-nand x) env)
           (expr-eval x env))
  :hints (("goal" :induct (induct-convert-to-nand x env)
                  :do-not-induct t)))
```

- Congruence
  - Allows the prover to treat certain equivalence-predicates as equalities. Special type of rewrite.

```
(defthm set-equiv-implies
    (implies
     (set-equiv! x y)
     (iff (member item x)
          (member item y)))
  :rule-classes (:congruence)
  :hints (("goal" :use set-equiv!-necc)))
```

Rule Classes cont.

- Forward-Chaining
  - Adds information to the prover extending the information available during the proving process.

```
(defthm subset-implies
    (implies (and (subset x y)
                  (member a x))
             (member a y))
  :hints (("goal" :use subset-necc))
  :rule-classes (:rewrite
                 :forward-chaining))
```

- Linear
  - Provides rewrite rules for the prover to use during linear arithmetic.

```
(defthm member-atom-count
    (implies (member a l)
             (>= (atom-count a l) 1))
  :rule-classes (:linear :rewrite))
```

ACL2 Tools

- :monitor
  - Identify when and why a rule was, or was not being applied during a theorem
- (verify)
  - See what information has been added by forward-chaining.
  - Figure out which rules apply to different terms.
  - Figure out which hypothesis are not being satisfied.

- :useless-runes
  - I wrote several rules which proved to be useless in many cases.
  - By disabling useless rules I significantly increased the speed of execution.

Conclusion

- I found many parts of the project particularly challenging
  - Parsing failed proofs
  - Predicting necessary lemmas
- The discrete math course I had taken at coe was the most useful in understanding how to utilize ACL2.
- This project was not a formal educational experience in using ACL2
  - Lacked repetition in using some proof methods and terms
  - Many challenges came about at unexpected times
- Biggest Takeaways
  - Understanding simplification

Failed Proofs:

The failed proofs tend to be large strings of logic.

This is an error for a theorem that occurs when the theorem prover lacks information about the transitivity of subset.

In my experience, the difficulty of using ACL2 is knowing what information the theorem prover is lacking.

```
Subgoal *1/3.2''
(IMPLIES
 (AND
  (WEAK-NANDP EXPR)
  (DISJOINT
       VARLIST2
       (BOUND-VAR-FINDER (MV-NTH 0
                                 (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR1 EXPR)
                                                        VARLIST))))
  (NOT (SUBSET VARLIST2
               (MV-NTH 1
                       (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR1 EXPR)
                                              VARLIST))))
  (SUBSET VARLIST2 VARLIST)
  (MEMBER-EQUAL
   (DISJOINT-WITNESS
        VARLIST2
        (BOUND-VAR-FINDER
             (MV-NTH 0
                     (MAKE-UNIQUE-BOUND-VAR
                          (NAND->EXPR2 EXPR)
                          (MV-NTH 1
                                  (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR1 EXPR)
                                                         VARLIST))))))
   (BOUND-VAR-FINDER
    (MV-NTH
     0
     (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR2 EXPR)
                            (MV-NTH 1
                                    (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR1 EXPR)
                                                           VARLIST)))))))
 (NOT
  (MEMBER-EQUAL
     (DISJOINT-WITNESS
          VARLIST2
          (BOUND-VAR-FINDER
               (MV-NTH 0
                       (MAKE-UNIQUE-BOUND-VAR
                            (NAND->EXPR2 EXPR)
                            (MV-NTH 1
                                    (MAKE-UNIQUE-BOUND-VAR (NAND->EXPR1 EXPR)
                                                           VARLIST))))))
     VARLIST2))).
```

Conclusion

- I found many parts of the project particularly challenging
  - Parsing failed proofs
  - Predicting necessary lemmas
- The discrete math course I had taken at coe was the most useful in understanding how to utilize ACL2.
- This project was not a formal educational experience in using ACL2
  - Lacked repetition in using some proof methods and terms
  - Many challenges came about at unexpected times
- Biggest Takeaways
  - Understanding simplification

Enhancements?

```
:hints ((pattern::hint
         (:and
          (not (subset x y)))
         :in-theory (enable subset)
         :restrict ((subset ((x x) (y y)))))))
```

```
(pattern::hint
 (not (subset x y))
 :expand! (subset x y))
```

- I found that I frequently found myself writing rules which “restricted” usage of those quantifiers whenever a negated instance of the quantifier was present.

- Ideally, I would just use :expand. However expand fails whenever the definitions are disabled.

```
(pattern::hint
 (not (subset x y))
 :expand (subset x y))
```

End of Presentation