HWK2 September 2007 Panagiotis Manolios
Due 10/17/2007 *before class starts.*

Exercise 1. Develop an *efficient* algorithm that given as input a Boolean formula over variables $x_1, x_2, \ldots, x_n$ and Boolean operators $\wedge, \vee, \neg, \Rightarrow$, generates an equivalent formula, but where $\neg$ is applied only to variables. For example, $\neg(x_1 \wedge (x_2 \vee \neg x_3))$ might get turned into $\neg x_1 \vee (\neg x_2 \wedge x_3)$. Note that you cannot introduce new variables. Analyze the running time of your algorithm, and, as precisely as you can, bound the size of the output in terms of the size of the input.

Exercise 2. Let f be a Boolean formula over variables $x_1, x_2, \ldots, x_n$ and Boolean operators $\wedge, \vee, \neg$. A subformula is an *even* formula if it falls under an even number of negations: that is, if we were to construct the parse tree, then the number of $\neg$ operators from this subformula to the root of the parse tree is even. If a subformula is not even, it is *odd*. Consider a modification to the Tseitin transformation, where instead of generating a new variable, say y, for subformula g and adding the constraint $y \Leftrightarrow g$, instead, we just add the constraint $y \Rightarrow g$, if g is even, and $g \Rightarrow y$ otherwise. Give pseudocode for the modified Tseitin and prove that the CNF generated by your code is equisatisfiable with f. Hint: First think about the restricted case where $\neg$ is only applied to variables.

Exercise 3. You want to check the satisfiability of two CNF formulas, f and g, *i.e.*, f and g are sets of clauses. There is substantial overlap between f and g. You will start by checking f, but when you check g, you would like to take advantage of all the work the SAT solver did to check f. Develop an algorithm to do this and argue that it is correct. How do you think your algorithm will perform and why?

Exercise 4. Come up with an interesting application for SAT. Perhaps it is based on work you are doing or perhaps it is related to a hard problem you encountered at some point in your career. Why is your application interesting? I encourage to come up with something cool. How can you encode it in CNF? Download at least two different SAT solvers, and use them to solve several hard instances of your problem. Explain your experimental results.

Exercise 5. Recall the definition of NICE dags. Let $VARS$ be a set of Boolean variables. Then a *Negation, Ite, Conjunction, and Equivalence dag (NICE dag)* over $VARS$ is a dag, $C = (V, E)$ such that

1. $E \subset V \times V$
2. Each vertex, $v \in V$, is labeled with an operator, $op(v) \in OPS = VARS \cup \{\neg, \leftrightarrow, \wedge, ite, true, false\}$.
3. $\langle \exists v \in V \text{ s.t. } op(v) \in \{true, false\} \rangle \Rightarrow V = \{v\}$.
4. Each vertex $v \in V$ s.t. $op(v) = \neg$ has an additional label, $arg(v) \in V$ s.t. $op(arg(v)) \neq \neg$.
5. Each vertex $v \in V$ s.t. $op(v) = ite$ has 3 additional labels, $test(v) \in V$, $then(v) \in V$, and $else(v) \in V$.
6. All $v \in V$ s.t. $op(v) \in \{\neg, \leftrightarrow, \wedge, ite\}$ are labeled with $args(v) \subseteq V$ such that
   - $args(v) = \{w \in V \mid (v, w) \in E\}$
   - $op(v) = \wedge \;\Rightarrow\; |args(v)| \geq 2$
   - $op(v) = ite \;\Rightarrow\; (args(v) = \{test(v), then(v), else(v)\} \wedge |args(v)| = 3 \wedge op(test(v)) \neq \neg \wedge op(then(v)) \neq \neg)$
   - $op(v) = \neg \;\Rightarrow\; args(v) = \{arg(v)\}$
   - $op(v) = \leftrightarrow \;\Rightarrow\; |args(v)| = 2 \wedge \langle \forall w \in args(v) :: op(w) \neq \neg \rangle$
   - $\langle \forall u, w \in V : op(u) = \neg \;\wedge\; w = arg(u) : u \notin args(v) \vee w \notin args(v) \rangle$
7. No two vertices have the same labels.
8. There is exactly one source in V, denoted $source(C)$

The semantics of NICE dags are fairly straightforward. Variables are given values by an environment and operators are applied to the values of their operands in the usual way.

Definition 1 *Given a NICE dag,* (V, E) *over variables* $VARS$*, we define an* evaluator function for NICE dags*,* $V \times (VARS \to \{true, false\}) \to \{true, false\}$ *as follows.*

- *When* $op(v) \in VARS$*,* $[\![v]\!]^\epsilon = \epsilon(v)$*.*
- *When* $op(v) = \wedge$*,* $[\![v]\!]^\epsilon = \bigwedge_{w \in args(v)} [\![w]\!]^\epsilon$*.*
- *When* $op(v) = \leftrightarrow$*,* $[\![v]\!]^\epsilon = v_1 \leftrightarrow v_2$ *where* $\{v_1, v_2\} = args(v)$*.*
- *When* $op(v) = ite$*,* $[\![v]\!]^\epsilon = \begin{cases} [\![then(v)]\!]^\epsilon & when\ [\![test(v)]\!]^\epsilon = true \\ [\![else(v)]\!]^\epsilon & otherwise \end{cases}$

We now define equivalence between two NICE dags as being the equivalence of their respective values under any environment.

Definition 2 *Two NICE dags,* C_1, C_2 *over variables* $VARS$*, are said to be* equivalent *if, for all* $\epsilon : VARS \to \{true, false\}$*,* $[\![source(C_1)]\!]^\epsilon = [\![source(C_2)]\!]^\epsilon$*. We denote this as* $C_1 \equiv C_2$*.*

NICE dags are built from the ground up, by using functions that create a new NICE dag by applying an operator to existing NICE dags. Some of the algorithms for doing this are given in Figure 1. To ensure the uniqueness of vertices, we keep a global table, $GTAB$ containing all the vertices that we create. We also maintain the invariant that $GTAB$ only contains NICE dags. Initially, $GTAB$ contains one vertex for each variable, as well as the nodes ZERO and ONE, which have *ops* *false* and *true*, respectively. Assume that we have functions `uand`, `unot`, and `uiff` that check if a node with the corresponding *op* and labels already exists in the $GTAB$. If it does, that node is returned. Otherwise, a new node with those labels is created, inserted into $GTAB$, and returned.

The algorithms in Figure 1 maintain the invariant that every vertex in $GTAB$ corresponds to a NICE dag over VARS. Each algorithm has the precondition that the input vertices are already in $GTAB$. The algorithms return a vertex corresponding to a NICE dag that is equivalent to the result of creating a NICE dag corresponding to applying the same operation to the inputs dags.

Only code for the operations `and` and `not` are provided. Your job is to provide code for `or`, `implies`, and `iff`. Also, provide the pre and post conditions for `iff`.

$\texttt{and}\,(W)$
Pre: $|W| \geq 2, \langle \forall v \in W :: v \in GTAB \rangle$
Post: $\langle \forall \epsilon :: [\![\texttt{and}\,(W)]\!]^{\epsilon} = \bigwedge_{v \in W} [\![v]\!]^{\epsilon} \rangle$
  $W' := W$
  if $\langle \exists v \in W' :: v = \text{ZERO} \rangle$ **then**
    return ZERO
  else
    if $\langle \exists v \in W' :: v = \text{ONE} \rangle$ **then**
      $W' := W' \backslash v.$
    if $W' = \{v\}$ **then**
      return v
    else if $\langle \exists v_1, v_2 \in W' :: v_1 = \texttt{not}\,(v_2) \rangle$ **then**
      return ZERO
    else
      return $\texttt{uand}\,(W')$)

$\texttt{not}\,(v)$
Pre: $v \in GTAB$
Post: $\langle \forall \epsilon :: [\![\texttt{not}\,(v)]\!]^{\epsilon} = \neg\, [\![v]\!]^{\epsilon} \rangle$
  if $v = \text{ONE}$ **then**
    return ZERO
  else if $v = \text{ZERO}$ **then**
    return ONE
  else if $op(v) = \neg$ **then**
    return $arg(v)$
  else
    return $\texttt{unot}\,(v)$

$\texttt{or}\,(W)$
Pre: $|W| \geq 2, \langle \forall v \in W :: v \in GTAB \rangle$
Post: $\langle \forall \epsilon :: [\![\texttt{or}\,(W)]\!]^{\epsilon} = \bigvee_{v \in W} [\![v]\!]^{\epsilon} \rangle$
  ???

$\texttt{impl}\,(v_1, v_2)$
Pre: $v_1, v_2 \in GTAB$
Post: $\langle \forall \epsilon :: [\![\texttt{impl}\,(v_1, v_2)]\!]^{\epsilon} = [\![v_1]\!]^{\epsilon} \Rightarrow [\![v_2]\!]^{\epsilon} \rangle$
  ???

$\texttt{iff}\,(v_1, v_2)$???

Figure 1: Code for `and` and `not`