Notes & Problem Set 3

DPL Seminar, Summer 2001
Handout 6

Konstantine Arkoudas

July 19, 2001
Problems

Due: Thursday, July 26.

Problem 1

In this problem you will write a function and a method for converting a proposition to CNF (conjunctive normal form). In what follows, by an atom we will understand a variable term such as ?A, ?v12, etc. We will only be concerned here with propositions that are built up from such atoms through the five propositional constructors: not, and, or, if, and iff.

A literal is either an atom or the negation of an atom. Thus ?A and (not ?A) are both literals; but (not (not ?A)) and (if ?A ?B) are not. We will use the letter $L$ to designate a typical literal. A proposition will be said to be in CNF iff it is of the form

$$P_1 \land P_2 \land \cdots P_n$$  \hspace{1cm} (1.1)

where each $P_i$ is a disjunction of literals:

$$L_1 \lor L_2 \lor \cdots L_k$$  \hspace{1cm} (1.2)

and $n, k > 0$. Association is immaterial both in 1.1 and in 1.2. Thus all of the following propositions will be considered to be in CNF:

$(\text{and } ?A \ (\text{and } (\text{not } ?A) \ ?C))$

and

$(\text{and } (\text{not } ?A) \ (\text{or } ?A2 \ (\text{or } ?A3 \ ?A4)) \ (\text{or } ?A5 \ (\text{not } ?A2)))$

and

$(\text{and } (\text{or } (\text{or } ?A2 \ ?A3) \ ?A4) \ (\text{or } ?A5 \ (\text{not } ?A2))) \ (\text{not } ?A))$

as well as ?B, (not ?A), and so on.

A proposition can be put into CNF by performing the following transformations:

1. Eliminate conditionals and biconditionals. This can be done by casting every biconditional $P \iff Q$ in the equivalent form $(P \Rightarrow Q) \land (Q \Rightarrow P)$, and every conditional $P \Rightarrow Q$ as $\lnot P \lor Q$. Thus by the end of this step, your proposition should only contain atoms, negations, conjunctions, and disjunctions.

2. Push all negation signs inward using De Morgan's laws:

$$\lnot (P \land Q) \iff (\lnot P \lor \lnot Q)$$

$$\lnot (P \lor Q) \iff (\lnot P \land \lnot Q)$$

3. Eliminate double negations. By the end of this step, your proposition should only contain conjunctions and disjunctions of literals.

4. Use the distributive laws

$$P \lor (Q \land R) \iff (P \lor Q) \land (P \lor C)$$

$$(P \land Q) \lor R \iff (P \lor R) \land (Q \lor R)$$

to push conjunctions out of disjunctions.
Part A
Write a function \texttt{cnf?} that takes a proposition \( P \) and returns \texttt{true} or \texttt{false} depending on whether or not \( P \) is in CNF.

Part B
Write a function \texttt{cnf-convert} that takes a proposition \( P \) and produces a proposition that is in CNF and logically equivalent to \( P \).

Part C
Write a method \texttt{cnf-derive} that takes a premise \( P \) and deduces from it a proposition that is in CNF and logically equivalent to \( P \).

Problem 2
Part A
(Note: The experiment described in this problem and the Athena code for it is due to Darko Marinov.)
First, issue the following declarations:

\begin{verbatim}
(domain Num)

(use-numerals (0 ...) Num)

(define-numeric-operations)
\end{verbatim}

Next, consider a binary function \texttt{make-and-tree} that takes an arbitrary proposition \( P \) and number \( n \) and builds a conjunction tree of height \( n \), with \( P \) at every leaf:

\begin{verbatim}
(define (make-and-tree leaf n)
  (match n
    (O leaf)
    (I (make-and-tree (and leaf leaf) (minus n 1))))))
\end{verbatim}

For instance,

\begin{verbatim}
> (make-and-tree true 3)
\end{verbatim}

Proposition: \( (\text{and} (\text{and} (\text{and} \text{true} \text{true})
                 (\text{true} \text{true}))
               (\text{and} (\text{and} \text{true}
                 (\text{true} \text{true})))
\end{verbatim}

Try constructing \( \text{make-and-tree} \text{ true} \text{ 16} \). Note the exponential increase in the size of the produced proposition as \( n \) varies. Now consider a method that takes a proposition \( P \) of the form \( \text{make-and-tree} \text{ true} \text{ n} \), for some \( n \), and derives \( P \). Here is one naïve way of writing such a method:

\begin{verbatim}
(define (naive P)
  (dmatch P
    (true (!claim P))
    ((and P1 P2) (!both (naive P1) (naive P2)))))
\end{verbatim}

Note that \texttt{naive} does the job, but is horribly inefficient. Try, for instance, \( \text{test naive n} \) for \( n = 15, \ldots, 20 \), and record the times it takes for each call. You should observe an exponential increase in the time series. (The function \texttt{test} is given below.)

Now consider an alternative way of doing this:
(define (trm P M)
  (dcheck
   ((holds? P) (M))
   (else (dmatch P
             ((and P1 P2)
               (!trm P1 (method ())
                (!trm P2 (method ())
                (dbegin
                 ((both P1 P2)
                  ((M))))))))))

(define (fast P)
  ((trm P (method () (!claim P))))

Try evaluating (test fast n) for n = 15, ..., 20; record the running times and compare them to the corresponding times for naive. (Note that (holds? true) always returns true, since, for convenience, the proposition true is a member of every assumption base.)

Explain the source of the disparity, in clear English and in as much detail as you can. Why is fast so much more efficient than naive? Can you explain the gist of the issue in terms of Athena’s formal semantics? Think of how the assumption base is threaded during the evaluation of a method call. To make things concrete, trace the evaluation of some simple method calls such as

(!fast (make-and-tree true 2)) or (!fast (make-and-tree true 3)) and observe the growth pattern of the assumption base. Do the same thing for naive and contrast.

Finally, here is the test function:

(define (test M n)
  (begin
   ((M (make-and-tree true n))
    (print "\nDone...\n")))

Part B

Write a unary method rassoc that takes a conjunctive premise P and deduces a conjunction P’ that is equivalent to P and right-associated. Thus, for instance, rassoc should be able to make the following derivations:

\[(\text{and} \ (\text{and} \ ?A \ ?B) \ ?C) \rightarrow (\text{and} \ ?A \ (\text{and} \ ?B \ ?C))\]

\[(\text{and} \ (\text{and} \ ?A \ (\text{and} \ ?E \ ?D)) \ (\text{and} \ ?C \ ?F)) \rightarrow (\text{and} \ ?A \ (\text{and} \ ?E \ (\text{and} \ ?D \ (\text{and} \ ?C \ ?F))))\]

\[(\text{and} \ ?A \ (\text{and} \ ?B \ ?C)) \rightarrow (\text{and} \ ?A \ (\text{and} \ ?B \ ?C))\]

\[?A \rightarrow ?A\]

In this problem we will only consider pure conjunctions (meaning that when the proposition is viewed as a tree, every internal node of it is an and), with variable atoms such as ?A and ?foo at the leaves. One possible solution is the following. First, we define a function get-leaves that returns a list of all and only the leaves of some given pure conjunction P:

(define (get-leaves P)
  (match P
          ((and P1 P2) (join (get-leaves P1) (get-leaves P2)))
          (l, [P])))

Next, we write a method detach that derives a leaf L from a pure conjunction P that is known to be in the assumption base. All we know about L is that it is a leaf of P. Since we do not know its exact location, we might have to search every branch of P in the worst case, deducing subtrees of P along the way:
(define (detach L P)
  (dmatch P
    ((and _ _) (try (!detach L (!left-and P)))
     (!detach L (!right-and P)))
    _ ((claim L))))

Finally, we define a method conjoin that takes a pure conjunction P known to be in the assumption base, and a list L consisting of one or more leaves of P, and derives a right-associated conjunction consisting of the leaves in L:

(define (conjoin P leaves)
  (dmatch leaves
    ([L] (!detach L P))
    ((list-of L rest) (!both (!detach L P) (!conjoin P rest)))))

The desired method can now be expressed as:

(define (transoc P)
  (!conjoin P (get-leaves P)))

Explain why this solution is inefficient, and give an alternative, more efficient solution. (Note that this is related to issues brought up in the previous problem.) Prove that your solution is more efficient: argue why it should be more efficient, and demonstrate that it is by presenting the running times of various test cases for both versions.

Problem 3

In this and the next problem we will continue to study the simple functional language L that was introduced in the previous problem set. Recall that an expression E generated by the abstract grammar of L is essentially a labelled tree. For instance, \( \lambda x.y \) can be thought of as the three-node binary tree with the keyword \( \lambda \) at the root, the variable \( x \) as the left single-node child, and the variable \( y \) as a similar child on the right side. Now in any tree structure \( T \), we may represent the position of any node in \( T \) as a list of integers, such as \([2,1,2]\), that represents the path that one must traverse in order to get from the root to the node in question. The position of the root node, in particular, is always the empty list \([\,]\). Assuming that we are ordering children from left to right, the position of the first child of the root is \([1]\); the position of the second child of the root is \([2]\); and so on. The position of the third child of the first child of the root is \([1,3]\), which can be read as: “Travel down the first branch of the root to get to the root’s first child, then travel down the third branch from there.” The set of all valid positions for a given tree \( T \) is often called the tree domain of \( T \), written as \( dom(T) \).

Part A

Write a unary function domain that takes an expression \( E \) (represented as a term of sort Exp) and returns a list of lists of numbers (terms of sort Num) representing the set of positions of \( E \), viewing the latter as a tree. For instance, \( \text{domain (LamExp 'x (VarExp 'x))} \) should return \([[],[1],[2]]\); and

\( \text{domain (CondExp true 1 (LamExp 'x (VarExp 'x))}) \)

should return \([[],[1],[2],[3],[3,1],[3,2]]\).
Part B

Write a binary function \texttt{label} that takes an expression \( E \) and a position \( p \) and returns the “label” that occurs in \( E \) at position \( p \). The label will be either an internal constructor such as \texttt{LamExp} or a leaf, namely, a numeral or an identifier. E.g., \texttt{(label (PairExp 37 'foo) [])} should return \texttt{PairExp}, while \texttt{(label (PairExp 37 'foo) [2])} should return 'foo. Your function should generate an error if the given position is not valid (i.e., not in \texttt{dom}(E)).

Problem 4

In this problem we will study and implement the formal semantics of \( \mathcal{L} \). The main point of this exercise (and its upcoming successors) is to illustrate the use of deductive systems in programming language theory, and in particular the use of Athena as a logical framework for implementing such systems.

Variable binding and substitutions

A very important notion in this semantics (and one that resurfaces in many other diverse contexts) is the operation of substitution, where every free occurrence of an identifier \( I \) within some expression \( E \) is replaced by some other expression \( E' \). The result of this operation is often denoted by \( E[E'/I] \). We call \( E \) the “base” expression and \( E' \) the “replacement”. Any substitution operation in a language that is rich enough to have variable-binding constructs (such as \( \lambda \) quantifiers, \texttt{let}, the sum operator \( \sum \) or integrals \( \int \) in mathematics, etc.) always carries with it the risk of \textit{variable capture}, whereby some free identifier occurrences of the replacement \( E' \) are captured by some binders of the base expression \( E \). Why is this undesirable? Because, intuitively, we want \( E[E'/I] \) to “say the same thing” about \( E' \) that it says about \( I \); but if some free variables of \( E' \) become bound in the process, this will not be the case. For instance, suppose that the base is the proposition \( (\exists y) x \neq y \), the variable\(^1\) whose free occurrences we want to replace is \( y \), and the replacement is \( y \). Then carrying out the substitution will result in \( (\exists y) y \neq y \), which clearly says something different than the original proposition!

Fortunately, there is a way to avoid this pitfall: before we carry out the replacement, we first consistently rename all the bound variables of the base \( E \) to be distinct from the variables of the replacement \( E' \); variable capture then becomes impossible. In the above example, we could rename the base to \( (\exists z) x \neq z \), since \( z \) does not occur in the replacement \( y \), and then safely carry out the substitution, producing \( (\exists z) y \neq z \). This is a safe thing to do because, in the vast majority of cases,\(^2\) consistently renaming the bound variables of an expression does not change the expression’s \textit{meaning}. Thus, for instance, \((\exists y) x \neq y \) and \((\exists z) x \neq z \) make the exact same assertion—that there is an object which is distinct from \( x \). We say that expressions which differ only in the names of their bound variables are \textit{alphabetically equivalent}, or \textit{algebraic variants}. So, in lexically scoped languages, alphabetic variants are semantically equivalent, and are customarily treated as identical. For example, we do not bother to distinguish between

\[
\sum_{i=0}^{\infty} f(i) \quad \text{and} \quad \sum_{j=0}^{\infty} f(j). \]

\(^1\)In this discussion we are using “variable” and “identifier” interchangeably, although in certain other contexts we may want to distinguish between the two.

\(^2\)More precisely, in languages that are \textit{lexically scoped}, where the \textit{scope} of a bound variable is clear by simple inspection of the surrounding text. By contrast, in dynamically scoped languages such as early LISP versions, the meaning of an expression might not necessarily be invariant under simple alphabetic renamings.
In $L$, the situation is fairly simple; there are only three variable-binding syntactic constructors: \textbf{let}, \textbf{fix}, and the all-powerful $\lambda$. Specifically, in an expression of the form $\lambda I . E$, the $\lambda$ binds $I$ within $E$ (and thus we say that $E$ represents the “scope” of $I$); in an expression of the form $\textbf{let} \ I = E_1 \ \textbf{in} \ E_2$, the \textbf{let} binds $I$ within $E_2$ (but not within $E_1$—thus the scope of $I$ consists only of $E_2$); and in an expression of the form $\textbf{fix} \ I . E$, the \textbf{fix} constructor binds $I$ within $E$, which thus comprises the scope of $I$.

More precisely, we can represent an \textit{occurrence} of an identifier $I$ within an expression $E$ as a pair $(I, p)$ consisting of $I$ and a position $p$ in $E$’s domain, viewing $E$ as a tree in the manner discussed in Problem 3. For instance, in the expression $(x, \lambda x. \varpi)$, the underlined occurrence of $x$ can be represented as $(x, [1])$, while the overlined occurrence is $(x, [2, 2])$; the unmarked occurrence is $(x, [1, 1])$.

With this representation, we can define an occurrence $(I, p)$ of an identifier $I$ in an expression $E$ to be \textit{bound} iff there is a position $q \in \text{dom}(E)$ such that $q$ is a prefix of $p$; the label that appears at position $q$ is either $\lambda$, or \textbf{let}, or \textbf{fix}; and the label that appears at position $q \oplus [1]$ is $I$, where $\oplus$ denotes list concatenation. We say that $(I, p)$ is \textit{free} iff it is not bound.

Accordingly, the set of identifiers that have free occurrences in an expression $E$ can be defined as follows:

\[
F(I) = \{I\} \\
F(n) = \emptyset \\
F(\textbf{true}) = \emptyset \\
F(\textbf{false}) = \emptyset \\
F(\langle E_1, E_2 \rangle) = F(E_1) \cup F(E_2) \\
F(\text{if}(E)) = F(E) \\
F(\text{fix} \ I . E) = F(E) - \{I\} \\
F(E_1 \ E_2) = F(E_1) \cup F(E_2) \\
F(\text{if} \ E_1 \ \text{then} \ E_2 \ \text{else} \ E_3) = F(E_1) \cup F(E_2) \cup F(E_3) \\
F(\textbf{let} \ I = E_1 \ \textbf{in} \ E_2) = F(E_1) \cup [F(E_2) - \{I\}] \\
\]

An expression $E$ is called \textit{closed} iff $F(E) = \emptyset$. Expressions that are not closed are called \textit{open}.

\textbf{Formal semantics}

First, we single out a certain subset of expressions $E$ that we will call \textit{values} $V$. Intuitively, values are expressions in “normal form”: for our purposes, they cannot be simplified any further. In standard BNF-grammar style, the definition of values is as follows:

\[
V ::= n | \text{true} | \text{false} | \lambda I . E | \langle V_1, V_2 \rangle \\
\]

Thus an expression is a value iff it is a numeral, or one of the two boolean constants \textbf{true} and \textbf{false}, or a $\lambda$-abstraction, or a pair consisting of values. This definition of values could also be cast as a deductive system, in terms of the following inference rules establishing judgments of the form $\vdash E : \text{Val}$ (read as “$E$ is a value”):

\[\text{E}\]
The semantics of \( \mathcal{L} \) are given by judgments of the form \( \vdash E \rightarrow V \), to be read “\( E \) evaluates to \( V \)”, or simply “the value of \( E \) is \( V \)”. The inference rules for deriving such judgments are shown in Figure 1.1. Note that rules [R8] and [R10] determine a call-by-value semantics: the argument of a function call must be fully evaluated before it is substituted in place of the formal parameter.

**Part A**

Write a binary function \texttt{free-occurrence}? that takes an expression \( E \) and an occurrence of a variable \( I \) in \( E \) and returns \texttt{true} or \texttt{false} according to whether or not that occurrence is free or bound, respectively. Next, write a unary function \texttt{bound-occurrences} that takes an expression \( E \) and returns a list of all and only those identifier occurrences in \( E \) that are bound. Finally, write a unary function \texttt{fvar} that takes an expression \( E \) and computes a list containing all and only those identifiers that occur free in \( E \).

**Part B**

Implement a ternary substitution function \texttt{sub} that takes a base expression \( E \), an identifier \( I \), and a replacement expression \( E' \) and returns \( E[E'/I] \). Your function should be safe: it should preclude variable capture.

**Part C**

Write an interpreter \texttt{eval} that takes a closed expression \( E \) and produces a value \( V \) iff \( \vdash E \rightarrow V \). Note that for one direction of this “iff” you will have to assume a so-called “value uniqueness” theorem: that every expression has at most one normal form, or more formally, that if \( \vdash E \rightarrow V_1 \) and \( \vdash E \rightarrow V_2 \) then \( V_1 = V_2 \). (Can you prove that informally? How would you go about that?)