Notes & Problem Set 4

DPL Seminar, Summer 2001
Handout 8

Konstantine Arkoudas

August 2, 2001
Problems

Due: Thursday, August 9, 2001.

Problem 1

Recall that every Athena variable $x$ that occurs free in a proposition $P$ has a most general sort in $P$. This is denoted by $\text{mgs}(x,P)$. For instance,

$$\text{mgs}(?L, (= ?L \text{ (Cons (Cons true Nil) Nil)))} = \text{(List-Of (List-Of Boolean))}$$

while

$$\text{mgs}(?L, (= ?L \text{ (Cons (Cons Nil Nil) Nil)))} = \text{(List-Of (List-Of (List-Of #T)))}.$$ 

for a sort variable #T.

An assumption base $\beta = \{P_1, \ldots, P_n\}$ is called uniform iff for every variable $x$ that occurs free in $\beta$, the sorts

$$\text{mgs}(x,P_1), \ldots, \text{mgs}(x,P_n)$$

are unifiable. Intuitively, two or more sorts are unifiable iff they are compatible with one another. More precisely, we say that sorts $S_1, \ldots, S_n$ are unifiable iff there is a substitution $\theta$ from sort variables to sorts such that if we apply $\theta$ to each $S_i$ we obtain identical results (i.e., $\theta$ “unifies” $S_1, \ldots, S_n$). For instance, (List-Of #T) and (List-Of Boolean) are unifiable because the substitution $\{#T \mapsto \text{Boolean}\}$ collapses them to the common instance (List-Of Boolean).

Athena requires assumption bases to be uniform and enforces this condition dynamically, at runtime. For instance, every time a proposition $P$ with some free variable $x$ is inserted in the assumption base $\beta$, Athena ensures that the most general sort of $x$ in $P$ is unifiable with the most general sorts of $x$ in those propositions in $\beta$ that have free occurrences of $x$. An error is generated if that is not the case. Explain why this is necessary by adding a deduction which would derive a contradiction from a consistent assumption base if this provision were not enforced.

Problem 2

In this problem we will axiomatize the semantics of $\mathcal{L}$. This will enable us, for instance, to prove that two expressions produce the same value, or that an expression diverges (gets into an infinite loop). First declare a unary relation symbol is-value as follows:

(declare is-value -> (Exp Boolean))

Axiomatize this relation using the specification of values in the third problem set, so that

(is-value $E$)

holds iff $E$ is a value. Then introduce a binary relation \texttt{evals} with the following signature:

(declare evals -> (Exp Exp Boolean))

Axiomatize this relation using the formal semantics of $\mathcal{L}$ that were given in the third problem set, so that

(evals $E_1$ $E_2$) holds iff $\vdash E_1 \sim E_2$.

(Thus \texttt{evals} is intended to capture precisely the relation $\sim$.)

Note that you will need to deal with substitutions deductively. For instance, you will probably need to introduce and axiomatize predicates such as \texttt{occurs-free} and \texttt{sub}, where (occurs-free $I$ $E$) holds iff the identifier $I$ occurs free in $E$, and (sub $E_b$ $I$ $E_r$ $E$) holds iff $E$ can be obtained by safely replacing every free occurrence of $I$ within $E_b$ by $E_r$. 

1
Part A
Prove the following proposition:

\[
\begin{align*}
&\text{(forall } ?E1 \\
&\quad \text{(forall } ?E2 \\
&\quad \quad \text{(if (evals } ?E1 ?E2) \\
&\quad \quad \quad \quad \text{(is-value } ?E2)))))
\end{align*}
\]

Part B
Prove the value-uniqueness theorem for $L$:

\[
\begin{align*}
&\text{(forall } ?E \\
&\quad \text{(forall } ?V1 \\
&\quad \quad \text{(forall } ?V2 \\
&\quad \quad \quad \text{(if (and (evals } ?E ?V1) \\
&\quad \quad \quad \quad \text{(evals } ?E ?V2)) \\
&\quad \quad \quad \quad \quad (= ?V1 ?V2))))))
\end{align*}
\]

Part C
Based on your formalization, write an interpreter for $L$ as a method. In other words, write a unary method `deval` which takes an expression $E$ and derives a theorem of the form $(\text{evals } E V)$.

Part D
Formalize `is-value` and `evals` using primitive methods rather than axioms. This consists in introducing a primitive method for each of the rules of the formal semantics of $L$. For instance, the rule

\[
\begin{align*}
&\vdash E \leadsto (V_1, V_2) \\
&\vdash l(E) \leadsto V_1 \\
&[R_5]
\end{align*}
\]

can be modeled by the following primitive method:

\[
\begin{align*}
&\text{(primitive-method (left-rule premise) \\
&\quad \text{(match premise} \\
&\quad \quad ((\text{evals } E \text{ (PairExp } V1 _)) \text{ (check } (\text{holds? premise}) \text{ (evals } (\text{LeftExp } E) V1)) \\
&\quad \quad \quad \quad \text{(else (error "Invalid application of left-rule.")))))
\end{align*}
\]

Functions dealing with free variables and substitutions may be used within the body of these primitive methods.

Rewrite the interpreter `deval` to use this formalization of the semantics. Is this version of `deval` easier to write, and if so, why? Is there anything we can prove with one formalization that we cannot with the other?

Problem 3
Recall that a CNF proposition is of the form

\[
C_1 \land C_2 \land \cdots C_n
\]

(1.1)
where each \( C_i \) is a disjunction of literals \( L_{i1} \lor \cdots \lor L_{ik_i} \). We use the letters \( K, L, \) and \( M \) to designate literals. It is customary to represent such a disjunction \( C_i \) simply by the set of literals

\[
C_i = \{ L_{i1}, \ldots, L_{ik_i} \}
\]  

(1.2)

that occur in it. Of course repetitions do not count in sets, and this means that duplicate occurrences of the same literal are discarded in this representation. This is justified because disjunction is idempotent, meaning that \( P \lor P \equiv P \) for all \( P \). A set of literals of the form 1.2 is customarily called a clause. The empty clause \( \{ \} \) is considered to be tantamount to **false**. This is reasonable because, in general, a disjunction \( P_1 \lor \cdots \lor P_m \) is true if at least one of the alternatives holds; and clearly this could never be the case if there are no alternatives, i.e., if \( m = 0 \). The empty clause is often denoted by the symbol \( \square \).

The entire proposition 1.1 is represented as the set of clauses \( \{ C_1, \ldots, C_n \} \), namely the set of sets

\[
\{ \{ L_{11}, \ldots, L_{k_{11}} \}, \ldots, \{ L_{1n}, \ldots, L_{kn} \} \}.
\]  

(1.3)

For any proposition \( P \), we write \( \bar{P} \) for its clausal CNF representation; i.e., \( \bar{P} \) is the set of clauses (of the form 1.3) that we obtain by putting \( P \) in CNF form. For example, if \( P = A \Rightarrow (B \land C) \) then

\[
\bar{P} = \{ \{ \neg A, B \}, \{ \neg A, C \} \}.
\]

Resolution is a refutation technique that aims to show that a proposition \( P \) is unsatisfiable by deriving a contradiction from it. Resolution operates on CNF propositions (which in this problem will be represented by lists). The technique is based on the **resolution rule**, which eliminates complementary literals from two clauses. (The complement of a literal \( L \) is defined as \( L' = \neg A \) if \( L \) is an atom \( A \), and \( L' = A \) if \( L \) is an atom negation \( \neg A \). Two literals are called **complementary** iff each is the complement of the other.) Schematically, the resolution rule can be depicted as

\[
\frac{\{ L, K_1, \ldots, K_m \}, \{ L', M_1, \ldots, M_n \}}{\{ K_1, \ldots, K_m, M_1, \ldots, M_n \}} \quad [\text{Resolution}]
\]

Thus applying the resolution rule consists in “striking out” the complementary literals \( L \) and \( L' \). The resulting clause is called the **resolvent** of the two input clauses. For instance, applying the rule to the two clauses \( \{ A, B, C \} \) and \( \{ A, \neg B, D \} \) strikes out \( B \) and \( \neg B \), resulting in the resolvent \( \{ A, C, D \} \). We say that a pair of clauses are **resolvable** iff they contain complementary literals. Note that two resolvable clauses may contain several pairs of complementary literals, and in that case there are multiple resolvents depending on which literals we choose to eliminate. Consider, for instance, the clauses \( \{ \neg A_1, A_2, A_3 \} \) and \( \{ \neg A_2, A_1 \} \). Here we could either choose to resolve \( \neg A_1 \) and \( A_1 \), obtaining the resolvent \( \{ A_2, A_3, \neg A_3 \} \); or we could choose to strike out \( A_3 \) and \( \neg A_3 \), obtaining the resolvent \( \{ \neg A_1, A_2, A_1 \} \). Formally, we define a resolvent of two clauses \( C_1 \) and \( C_2 \) as any clause \( C \) such that \( C = [C_1 \cup C_2] - \{ L, L' \} \) for some \( L \in C_1, L' \in C_2 \). Accordingly, the clauses \( \{ A, B \} \) and \( \{ B, C \} \) are not resolvable. Thus in this case the rule does not apply—there is no resolvent.

Bearing in mind that a clause \( \{ L_1, \ldots, L_k \} \) is intended to represent the disjunction \( L_1 \lor \cdots \lor L_k \); and that any proposition of the form \( P_1 \lor P_2 \) is equivalent to \( \neg P_1 \Rightarrow P_2 \); we see that an alternative but essentially equivalent way to cast the resolution rule is the following:

\[
\frac{\neg L \Rightarrow K_1 \lor \cdots \lor K_m \quad L \Rightarrow M_1 \lor \cdots \lor M_n}{K_1 \lor \cdots \lor K_m \lor M_1 \lor \cdots \lor M_n} \quad [\text{Resolution}]
\]
The soundness of this formulation is perhaps a bit more intuitive.

A resolution refutation attempts to derive the empty clause \( \Box \) (which amounts to the contradiction false, as explained above), by repeated applications of the resolution rule. Therefore, to refute \( P \), i.e., to show that \( P \) is unsatisfiable:

- Convert \( P \) to CNF, representing it as a set of clauses \( \tilde{P} \).
- While \( \Box \not\in \tilde{P} \) and \( \tilde{P} \) contains two resolvable clauses \( C_1 \) and \( C_2 \):
  - Add the resolvent of \( C_1 \) and \( C_2 \) to \( \tilde{P} \).
- If \( \Box \in \tilde{P} \) report \text{“}P \text{ is unsatisfiable”}, otherwise report \text{“}P \text{ is satisfiable”}.

Part A

Write a function get-cnf-clauses that takes a proposition \( P \) and converts it to CNF using the above clause representation. That is, the output of your function should be a list of clauses, where each clause is represented as a list of literals. Duplicate literal occurrences should be eliminated, and so should be duplicate clauses.

Part B

Define a function resolve that takes a proposition \( P \) and attempts to refute it using the foregoing algorithm. Thus the output of resolve must be either \text{“}unsatisfiable” or \text{“}satisfiable”. Your implementation should be functional, i.e., it should not use side effects, while loops, or begin expressions; and it should always terminate.

Part C

Any algorithm that can decide unsatisfiability can also decide validity, since a proposition \( P \) is valid iff \( \neg P \) is unsatisfiable. So we can use resolution to decide whether a proposition \( P \) is logically valid (a tautology) by using resolve on \( \neg P \). Further, any algorithm that can decide validity can also decide logical implication, since a proposition \( P \) is logically implied by a set of propositions \( \{P_1, \ldots, P_k\} \) iff the implication \( (P_1 \land \cdots \land P_k) \Rightarrow P \) is valid. In Athena this means that you can use resolution to find out whether an arbitrary proposition \( Q \) follows from an assumption base \( \beta \) by checking whether the proposition

\[
\bigwedge_{P \in \beta} P \Rightarrow Q
\]

is valid. Write a function follows? that takes an arbitrary proposition and produces true or false depending on whether or not it follows from the current assumption base.\(^1\)

Note that the assumption base can be accessed via the built-in unary function fetch-all that takes a proposition “tester” \( f \) (meaning that \( f \) is a unary function which takes an arbitrary proposition and returns either true or false) and produces a list of all and only those propositions \( P \) in the current assumption base that satisfy \( f \), i.e., such that \( f(P) = \text{true} \). Therefore, the call

\[
(\text{fetch-all } (\text{function } (P \text{ true}))
\]

will return a list of all the elements of the assumption base.

\(^1\) Recall that we are not talking about the full Athena logic here, but only about its propositional subset.
Part D

[Hard] Prove that the resolution algorithm given above is sound and complete. Soundness means that if the algorithm derives the empty clause from \( \overline{P} \) then \( P \) is indeed unsatisfiable. Completeness means that if \( P \) is unsatisfiable then the resolution algorithm will derive the empty clause from \( \overline{P} \).

Problem 4

A proposition \( P \) is said to be in \textit{prenex conjunctive normal form} (PCNF) iff it is of the form

\[
(Q_1 \; x_1) \cdots (Q_n \; x_n) \; P'
\]

where each \( Q_i \) is a quantifier (universal or existential) and \( P' \) is a quantifier-free proposition in CNF. We refer to the sequence \((Q_1 \; x_1) \cdots (Q_n \; x_n)\) as the \textit{prefix} and to the body \( P' \) as the \textit{matrix} of \( P \), respectively. Every proposition \( P \) can be transformed into an \textit{equivalent} proposition in PCNF by using the following rules:

1. Rename \( P \) so that no variable is bound by more than one quantifier.
2. Eliminate conditionals and biconditionals (this should be familiar from the CNF problem).
4. Pull quantifiers out of the matrix by using the equivalences

   \[
   \left[ (Q \; x) \; P_1 \right] \oplus P_2 \iff (Q \; x) \; [P_1 \oplus P_2] \\
   P_2 \oplus \left[ (Q \; x) \; P_1 \right] \iff (Q \; x) \; [P_2 \oplus P_1]
   \]

   which are valid whenever \( x \) does not occur free in \( P_2 \) (this will presumably be ensured by the first step), and for \( Q \in \{\forall, \exists\}, \oplus \in \{\land, \lor\} \).
5. Apply the distributive laws to the matrix:

   \[
   P_1 \lor (P_2 \land P_3) \iff (P_1 \lor P_2) \land (P_1 \lor P_3) \\
   (P_1 \land P_2) \lor P_3 \iff (P_1 \lor P_3) \land (P_2 \lor P_3)
   \]

Part A

The inference rule schema

\[
\begin{array}{c}
\left[ (Q \; x) \; P_1 \right] \oplus P_2 \\
Q \; x \; [P_1 \oplus P_2]
\end{array}
\]

provided \( x \) does not occur free in \( P_2 \)

gives rise to four distinct inference rules by choosing a value for \( Q \) from \( \{\forall, \exists\} \) and a value for \( \oplus \) from \( \{\land, \lor\} \). Write four methods to implement each such instantiation.

Part B

Write a unary function that converts a given proposition to a proposition in PCNF.

Part C

Write a unary method that takes any proposition \( P \) and derives a theorem of the form \( P \iff P' \), where \( P' \) is a proposition in PCNF.