Harvey Mudd College
Computer Science 80
Logic for Computer Science
Spring Semester 2002

Optional Extra-Credit Projects
Due 5:00pm, Thursday May 16, 2002

Optional Project 4 – First-Order Substitution and Unification

For this project you must implement the first-order substitution and unification algorithms. This will involve implementing five functions:

- `substituteInWFF`, which takes two arguments: a well-formed first-order formula and an assignment (which is a pair of a variable and a first-order term), and returns the result of subtituting the given term for all free occurences of the given variable in the given formula. This function depends on the following one to do some of its work.
- `substituteInAtom`, which takes two arguments: a first-order atom and an assignment, and returns the result of making the given substitution in the given atom.
- `substituteInTerm`, which takes two arguments: a first-order term and an assignment, and returns the result of making the given substitution in the given term.
- `unifyAtoms`, which takes two first-order atomic formulas as arguments and returns a most general unifying substitution (i.e. a list of assignments) if one exists. It depends on the following function for much of its work.
- `unifyTerms`, which takes two first-order terms as arguments and returns a most general unifying substitution if one exists.

Note that, in the last two functions, one must distinguish between success and failure, and, in the latter case, return a substitution as well. This is accomplished using `option` types, as discussed below.

During substitution into a formula, you will need in certain cicumstances, to generate a new variable not already in use. The easiest way to do this is to create a global counter and append it's value to a string such as `"x_"`, as in `x_327`. You may assume that any variable name of that form will be unique (i.e., none of that form will occur in your input).

SML Particulars

Representing Terms and Formulas

In SML you will represent term and formula structures using the data structures defined below.

```
type const = string         type var   = string
type func  = string         type pred  = string

datatype term = Const of const
              | Var of const
              | Term of func * term list;

datatype fowff = Bot
               | Top
               | Atom    of pred * term list
               | Not     of fowff
               | And     of fowff * fowff
               | Or      of fowff * fowff
               | Implies of fowff * fowff
               | Equiv   of fowff * fowff
               | Forall  of var * fowff
               | Exists  of var * fowff;
```

An assignment is just a pair of a variable and the assigned value:

```
type assignment   = (var * term);
```

The types of the subtitution functions should then be:

```
val substituteInTerm = fn : term  -> assignment -> term
val substituteInAtom = fn : fowff -> assignment -> fowff
val substituteInWff  = fn : fowff -> assignment -> fowff
```

Note that we won't actually store function symbol arity. We will assume that each function symbol can occur at each arity (though it is technically a different symbol at each arity). Of course this means that when you attempt to unify two terms with the same function symbol you must make sure that they are each actually being used at the same arity. The same caveat applies to the predicate symbols of atomic formulas.

Representing Unification Success and Failure

In SML you will use an option type to represent success and failure of the unification functions. If the unification algorithm fails, you will return `NONE`. If it succeeds you return `SOME` substitution, where a substitution is just a list of assignments. So, the type of the unification functions should be:

```
type substitution = assignment list;

val unifyTerms = fn : term  -> term  -> substitution option
val unifyAtoms = fn : fowff -> fowff -> substitution option
```

Optional Project 5 – First-Order Conversion to Clausal Form

For this project you must implement the algorithm which, given a first-order formula, returns the satisfiability-equivalent clausal formula.

You should define the project in terms of the data structures given above.

The project should implement a function named `clausal`, which takes a first-order well-formed formula as an argument and returns a list of lists of first-order literals representing the satisfiability-equivalent clausal formula.

As in project 4 above, you will need to use global counters to create new variables and skolem functions and constants.

While this project is technically discrete from the previous one, it will require that you implement substitution in order that you be able to rename variables and assign skolem constants and functions as necessary.

Optional Project 6 – First-Order Resolution

For this project you must implement first-order resolution.

You should define the project in terms of the data structures given above.

By analogy to the last required project, this should be implemented as two principal functions: `resolve`, which takes a pair of clauses and builds all possible resolvents, and `refute`, which runs the overall algorithm.