Harvey Mudd College
Computer Science 80
Logic for Computer Science
Fall Semester 1999

Optional Extra-Credit Projects
Due 5:00pm, Tuesday December 14, 1999

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 four functions:

- `substituteInWFF`, which takes three arguments: a well-formed first-order formula, 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.
- `substituteInTerm`, which takes three arguments: a first-order term, a variable, and a first-order term, and returns the result of subtituting the second term for all occurences of the given variable in the first term.
- `unifyAtoms`, which takes two first-order atomic formulas as arguments and returns a most general unifying substitution (i.e. a list of substitutions) 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 will be accomplished in different ways depending on the implementation language, 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).

Rex Particulars

Representing Terms

In Rex you will represent term and formula structures as lists, extending the representation used for propositional fomulas.

Terms wil be represented as follows:

Constants – `["Const",` *const name*`]`

Variables – `["Var",` *var name*`]`

Compound Terms – `["Term",` $func\ name$`,` $term_1$`, ...,` $term_n$`]`

Note, the last means 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 one 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 will apply to atomic formulas below.

Constant, variable, and function names are just Rex strings.

Representing Formulas Well-formed formulas will be represented as follows:

Atoms –

$\bot$ – `"FALSE"`

$\top$ – `"TRUE"`

$pred(t_1, \ldots, t_n)$ – `["Atom",` $pred\ name$`,` $term_1$`, ...,` $term_n$`]`

Non-Atomic WFFS – We build up WFFs from these using the same constructions as for the propositional resolution projects, augmented to handle quantified formulas:

$\neg A$ – `["NOT",`A`]`

$A \wedge B$ – `[`A`,"AND",`B`]`

$A \vee B$ – `[`A`,"OR",`B`]`

$A \Rightarrow B$ – `[`A`,"IMPLIES",`B`]`

$A \equiv B$ – `[`A`,"EQUIV",`B`]`

$\forall x(A)$ – `["FORALL", ["Var",` *var name*`],` A`]`

$\exists x(A)$ – `["EXISTS", ["Var",` *var name*`],` A`]`

Representing Unification Success and Failure

You will represent failure of the unification functions by returning 0 (zero). If unification succeeds then you should return a list of assignments. Each assignment is represented as a two element list with the variable as the first element and the assigned term as the second element.

We use 0 as the failure value rather than `[]` because it is possible for unification to suceed but return an empty substitution, as when we unify a constant with itself.

SML Particulars

Representing Terms and Formulas

In SML you will represent term and formula structures using the data structures defined below. These are analogous to the Rex specifications given above, and you should read that section for some caveats.

```
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;
```

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 a list of pairs. The first element of each pair is the variable, and the second is the assigned value. That is, the type of the unification functions should be:

```
val unifyTerms = fn : term -> term -> (var * term) list option
val unifyAtoms = fn : fowff -> fowff -> (var * term) list option
```

The types of the subtitution functions should be:

```
val substituteInTerm = fn : term -> var -> term -> term
val substituteInAtom = fn : fowff -> var -> term -> fowff
```

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 1 above, you will need to use global counters to create new variables and skolem functions and constants.

While this formula 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 runst the overall algorithm.