TP 2

01/12/2009

1 Leftovers

Exercise 1 (Programming with naturals).

1. We define the terms

$$\begin{array}{lcl} c_0 & = & \lambda s.\lambda z.z \\ c_1 & = & \lambda s.\lambda z.(s\ z) \\ c_2 & = & \lambda s.\lambda z.(s\ (s\ z)) \\ \ldots & & \end{array}$$

   which will represent the natural numbers. Define the term *succ* such that

$$\mathit{succ}\, c_i \rightarrow^* c_{i+1}.$$

2. Define the term *plus* such that

$$\mathit{plus}\, c_i\ c_j \rightarrow c_{i+j}.$$

3. Define the term *mult* such that

$$\mathit{mult}\, c_i\ c_j \rightarrow c_{i\times j}.$$

4. Define the term *iszero* which returns *true* if its argument is c_0 and *false* otherwise.

5. Define the term *pred* which returns c_0 if its argument is c_0 and c_{i-1} if its argument is c_i for some $i > 0$.

Exercise 2 (Emulating recursive functions).

1. A lambda-term with no free variables is called a *combinator*. One of the most "famous" combinators is:

$$Y = \lambda f.(\lambda x.(f\ (x\ x))\ (\lambda x.f\ (x\ x)))$$

   which is a fixpoint operator (see next). Prove that

$$Y\ g = g\ (Y\ g).$$

2. Define a lambda-term *fact* which returns the factorial of its argument.
3. Define a lambda-term *isprime* which returns true iff its argument is a prime number.
4. Explain why the Y-combinator cannot be used in a call-by-value setting.

2 First-Order Unification

Exercise 3.

We define an OCaml data structure for first-order terms as follows:

```
type term =
    Var of string
  | Func of string * term list;;
```

Then the term $f(x, x)$ is represented as:

```
let t = Func("f", [Var("x"); Var("x")]);;
```

1. Define an OCaml data structure for substitutions.
2. Write a function *unification* which takes two terms as arguments and returns either *None* (if the two terms do not unify) or *Some* σ (if the two terms unify and σ is their mgu). [you will probably need other helper functions as well]
3. Test your function and make sure it works. Ask your neighbors to provide examples that break your code. Provide examples to your neighbors that break their code. Then ask me to provide examples that break your code.
4. Create an example which proves that your code runs in exponential time. If you cannot find such an example due to the fact that no such example exists, kudos to you.
5. *(Optional).* Make your code run in polynomial time.
6. *(Optional).* Search on the Internet to find out how a (pure) Prolog interpretor works and write one.