Functional Programming
Carsten Schürmann
Date: October 2, 2002

Homework 3

Due: Monday, October 6, 2002.

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. None of the problems requires more than a few lines of code.

In this problem you are asked to implement a Hindley-Milner type reconstruction algorithm in SML. Consider the language of expressions that contains integers (addition, subtraction, multiplication), booleans (ifthenelse), functions (variables, application), lists (case), recursion, and Milner's let. The expressions that manipulate values of the respective types are given in parenthesis.

Exercise 1 Implement a datatype `Exp` with the appropriate constructors. (Hint: Due to inherent restrictions to SML, I would suggest to create an explicit constructor for variables. Consequently, you will need to implement substitution.)

Exercise 2 Implement an evaluator for your language.

Exercise 3 Implement a Hindley-Milner type inference algorithm for expressions.

```
infer  : Ctx * Exp -> Tp
check  : Ctx * (Exp * Tp) -> unit
```

Don't forget to distinguish between types `Tp` and type schemes `Ts`. You might want to consider implementating a unification algorithm to find the instantiation of type variables. `infer` infers a type in a context, and `check` checks if an expression has the right type.

Hints:

1. Use the code from class for unification.

```
whnf    : Tp -> Tp
occurs  : Tp option ref * Tp -> unit
occursW : Tp option ref * Tp -> unit
unify   : Tp * Tp -> unit
unifyW  : Tp * Tp -> unit
```

2. Before you add new declarations into the context, abstract away all free type variables. This makes it easier to use a polymorphic function. I suggest to store $\forall\alpha.\alpha \to \alpha$ as `All (Arrow (1, 1))`. The 1 is a reference to the appropriate universal binder. This way $\forall\alpha.\forall\beta.\alpha \to \alpha \to \beta$ is stored as `All (All (Arrow (1, Arrow (1, 2))))`.

```
collect      : Tp * Tp option ref list -> Tp option ref list
abstract     : Tp -> Ts
instantiate  : int *  Env * Ts -> Tp
```

   Organize this abstraction into three stages. The first stage is called collection where you collect all free variables into a list, the second, abstraction, where you compute a type scheme from the type, and finally, instantiation, where you instantiate the universal quantifiers by new ones. This comes in handy when you look up an object level variable in the context during type inference.