Type Reconstruction

João Paulo Pizani Flor   Liewe Thomas van Binsbergen

Friday, 12th April, 2013

1 Introduction

This was the fourth practical assignment of the course "Compiler Construction", given on the third academic period of 2012/2013. The goal of the assignment was to implement Hindley-Milner type inference (using Algorithm W). More specifically, the goal was to build a program which, given untyped lamda terms (extended with let bindings), would produce a SystemF term, with types annotations resulting from Hindley-Milner inference.

2 Compiling and running

The project is in "cabalized" form, but there is the addition of a pre-processing step, needed to generate Haskell files from the attribute grammars. So, to build the project completely, there is a Makefile available. To build the project, a simple `make` command in the project directory should be sufficient.

The build process results in three executables:

- `parse-hm` (parser of the untyped lambda calculus)
- `hm2systemf` (converter from HM term to SystemF term)
- `pp-systemf` (SystemF pretty-printer)

There are some test files under the "test" subdirectory in the project root. Each file contains an untyped HM term. Executing a test should generate the corresponding typed SystemF as output, giving some tracing information in the process.

Given the fact that it is necessary to execute a pipeline with 3 executables for each test, we made a little script (`test.sh` in the project root) to run tests. It should be run like follows, where `n` is the number of the desired test:

```
./test.sh <n>
```

3 General organization

All the code that effectively performs the HM to SystemF conversion is placed in the file `CCO/HM/AG/Base.ag`, together with the definitions of the HM grammar itself.

There is a typeclass called `Typelike`, which defines operations which are very similar for types, type schemes, type environments and type substitutions. These operations are how to obtain free type variables and how to apply a type substitution.

```
class Typelike a where
    ftv :: a -> S.Set TyVar
    (.$.) :: TySubst -> a -> a
```

As can be seen from the code above, we used Data.Set to store the free variables. Also, we used Data.Map to represent the type environment and type substitutions.

The algorithm W itself, as well as the conversion from HM untyped terms to SystemF terms, are done by using attributes, the algorithms are “embedded” in the attribute grammar.

4 Design decisions and some remarks

4.1 Source of fresh variables

One of the requirements for this project was to have a “source” of fresh variables, which is utilized in the various steps of Algorithm W. Initially, we thought about using a monad (Reader or Supply) for this task. We met, however, some difficulty while trying to integrate monadic code in the attribute grammar, and decided in the way to use an (infinite) list as supply and pass an attribute as book-keeping. Below there is the code responsible for obtaining fresh variables.

```
allTyVars :: [TyVar]
allTyVars = tyvars ++ concatMap ((\x -> map (++ x) tyvars) . show) [1..]
    where tyvars = ["a", "b", "c"]

nextTyVar :: [TyVar] -> TyVar
nextTyVar used = head $ nextTyVars used

nextTyVars :: [TyVar] -> [TyVar]
nextTyVars used = dropWhile (flip elem used) allTyVars
```

As can be seen, the function that generates fresh variables (`nextTyVars`) receives a list of *used* type variables as input, and then obtains the first variables which are *not yet* used.

We needed to keep track of type variable usage by introducing a *chained* attribute in the grammar, called “used”.

4.2 Attributes for Algorithm W and SystemF

As already mentioned, we implemented Algorithm W and the conversion to SystemF both inside the attribute grammar, so there are attributes that correspond to the inputs and outputs of these algorithms.

First of all, for Algorithm W: we needed an *inherited* attribute to pass around the type environment, called “env”. We also have a synthesized attribute for the value returned by Algorith W, called “res”. Finally, we added an attribute (synthesized) that represents the *principal type* of the term at each tree node, called “pri”.

For the conversion from HM to SystemF term there are two attributes at play. First, there is the synthesized attribute “f”, which is simply the SystemF version of the term at hand. Also, we needed an auxiliary inherited attribute, called “bds”, which stores type variables bound at let definitions.