A SYSTEM F IMPLEMENTATION IN AGDA

JAIME GUERRERO

1. Overview

This Agda implementation of System F is based upon Benjamin Pierce's implementation as described in *Types and Programming Langauges* and his website. This document outlines the basic components of and their interactions.

2. Types, Terms, and Bindings

Within System F, the basic objects of discourse are Types and Terms. The types are given by the grammar

$$\tau = Nat \mid Boolean \mid TypeVar\ i \mid \tau \Rightarrow \tau \mid Forall\ \tau \mid Empty$$

where $i \in \mathbb{N}$. Overall, these are typical types for System F, with the exception of *Empty*. It's inclusion is for convenience; there are situations where Agda's requirement for function totality calls for a type to represent an error. Any type that has *Empty* as a subtype will be considered invalid.

Terms are given by the grammar

$$e = Var\ i \mid Lam\ \tau\ e \mid App\ e\ e \mid TypeAbs\ e \mid TypeApp\ e \mid$$
$$Num\ i \mid Succ\ e \mid True \mid False \mid If\ e\ e\ e \mid Empty$$

where $i \in \mathbb{N}$. Again, the inclusion of *Empty* is a convenience; any term with *Empty* as a subterm is ill-formed.

This implementation uses de Bruijn indices for variables in relation to *Lam* tags, and type variables in relation to *TypeAbs* tags.

Bindings are generated by the following:

$$b = TypeVarBind \mid VarBind\ \tau \mid TyAbbBind\ \tau \mid TmAbbBind\ e\ \tau$$

TmAbbBind is used within both the evaluator and the type checker since it holds the expression that will replace a term variable, along with the expression's type. The other three binders are only used within the type-checker. Whenever a variable is encountered, its type is captured with a *VarBind*. A *TypeVarBind* is pushed onto the context upon finding a type variable; since a type variable does not have an explicit type bound to it, a *TypeVarBind* carries along no other information. A *TyAbbBind* holds a type that will replace a type variable. It's purpose is most clearly seen in the function `typeEq`, found in `TypeCheck.agda`. Whenever a type variable is encountered, its replacement type is pulled from the context.

Date: August 1, 2014.

3. Shifting

Given the above pieces of the language, shifting functions are defined in order to assist in evaluation. Since both types and terms have variable entities, both receive their own shifting functions.

For terms, the shifting function is `termShiftAbove`, which takes as arguments two natural numbers and a term to work upon. The first argument the the number by which each free variable in the term should be increased. The second argument is a "lower bound " of which free variables should be increased. For example, `termShiftAbove 12 1 (Lam Nat (App (Var 1) (Var 0)))` returns `Lam Nat (App (Var 1) (Var 0))`. Even though there is a free variable in `(Var 1)`, the second argument indicates that only free variables with index larger than 1 should have 12 added to them. In contrast, `termShiftAbove 12 1 (Lam Nat (App (Var 2) (Var 0)))` returns `Lam Nat (App (Var 14) (Var 0))`, since all free variables with index greater than 1 are incremented by 12. Also note that part of `termShiftAbove`'s functionality requires that its second argument be incremented by one each time it passes under a λ-abstraction. To see why, consider the fact that in a series of highly nested λ-abstractions, free variables get "further away". Incrementing this second argument can be thought of as maintaining the minimum index any free in the subterm might have. Shifting for types works in a similar manner.

In contrast to `termShiftAbove`, which increments free variables by a given value, `negTermShift` decrements free variables by a given amount. In Pierce's ML implementation, `termShiftAbove` can subtract from the indices of free variables by passing in negative integers. Since Agda's type system is much stricter, `negTermShift` was defined to handle subtraction and the complications associated with creating integers out of natural numbers.

4. Substitution

Substitution is built upon the shifting functions outlined above. The function `termSubst` takes a "lower bound" natural number (similar to `termShiftAbove`), a term that will be substituted, and a term upon which the substitution will take place. Again, the natural number argument is incremented when passing under a λ-abstraction. This time, it is in an attempt to maintain the distance of the free variable from the outermost λ-abstraction. For example, `termSubst 0 (Num 15) (Lam Nat (App (Var 0) (Var 1)))` produces `Lam Nat (App (Var 0) (Num 15))`. The initial argument of 0 is incremented to 1 when the function passes under the abstraction, and the substitution occurs on `(Var 1)`.

For constant terms with no free variables, blind substitution as outlined above works without difficulty. But in order to substitute arbitrary functions for variables, there needs to be a way to maintain the free and bound variables of the function. This notion is captured by the following Agda code:

```
termSubst j s (Var x) with (== x j)
... | true  = (termShift j s)
... | false = (Var x)
```

When the appropriate variable to be replaced is encountered, the call to `(termShift j s)` will ensure that all free variables within `s` remain free. Since `j` can be thought of as the number of binders that have been encountered so far, all free variables

within `s` need to be incremented by that many to remain free. As an example, consider `termSubst 0 (Lam Nat (App (Var 1) (Var 2))) (Lam Nat (Var 1))`, which returns `Lam Nat (Lam Nat (App (Var 2) (Var 3)))` and maintains the free variables in the second argument to `termSubst`.

Substitution at the type level is analogous.

Given a substituting term (type) `s` and a term (type) to work upon `t`, `termSubstTop` will replace the free variables within `t` that have the smallest index by `s`. This is given by the code

```
termSubstTop : Term -> Term -> Term
termSubstTop s t = negTermShift 1 (termSubst 0 (termShift 1 s) t)
```

In this situation, it is assumed that the first free variable of `t` has index 1. Since `s` will be inserted under at least 1 λ-abstraction, all of its free variables are incremented by 1 within the call to `termShift`. But within the call to `termSubst`, all free variables within `s` are incremented again by its own call to `termShift`. Finally, `negTermShift` decrements all variables by 1 and reflects the fact that `t` has one fewer free variable.

Substituting a type into a term proceeds by way of a call to `typeSubst` upon a λ-abstraction's type variable annotation.

5. Evaluation

Evaluation proceeds in a straightforward fashion. Because of the nameless representation of variables, getting a variable's value corresponds to indexing into the environment.Furthermore, the interpreter operates in a call-by-value manner by evaluating arguments to function applications as far as possible.

6. Type Checking

Type checking is also fairly standard. One requirement is that both branches of an `If` statement have the same type; this is to prevent evaluation of the test associated with the `If` statement. Also, although we assume well-typed, perfect inputs, the type checker will ensure that functions have arguments of the correct type. This is done by way of the `isTypeAligned` function, which ensure that functions of the type $\tau \Rightarrow \sigma$ are given arguments of type τ.