Experimentation Project

Ernesto Rodriguez

November 29, 2014

1 Introduction

This document describes the implementation of the analysis named: "Polyvariant Flow Analysis with Higher-ranked Polymorphic Types and Higher-order Effect Operators". This is a flow analysis for a higher order language which uses higher-ranked types to increase the presicion. The result of the analysis is delivered as an annotated type system for a simply typed lambda calculus. The major limitation of the analysis is that it does not support polymorphic types.

The analysis has been implemented in the Haskell programming language and also includes an interactive web interface that can be compiled using GHCJS [**?**]. Alternatively, the analysis can be compiled with the GHC compiler and the input is read form the standard input and the ouput is a Latex document with the results of the analysis.

2 File Structure

The code is divided in three modules: algorithms, types and web. The web group only contains details about manipulating dom elements and interfacing with the other two containers so it is not interesting and will not be explained any futher.

3 Types

This module defines all the types that were defined to model the analysis. This ammounts to:

- A type for the lambda calculus (LambdaCalc.hs)
- A type for simple types (Type.hs)
- A type for annotated types (AnnType.hs)
- A type for effects (Effect.hs)
- A type for annotations (Annotation.hs)
- A type for Sorts (Sorts.hs)

Additionally, the module contains the `Common` component which defines algorithms that operates in many of those types.

3.1 Traversals

Tree traversals over the mentioned types are defined as folds. The `Common` component provides a type class called `Fold` which defines the necessary methods for basic traversal over a structure. This includes a function to fold over the structure and two algebras: the regular algebra and the group algebra. The fold function is expected to provide a unique index to all elements of the structure and the functions of each of the algebras always take as a first argument that index.

Algebras are parametrized by three types: `alg m s r`. The `m` parameter is the monad under which the algebra operates. The `s` parameter is the type of the structure the algebra is meant to traverse and the `r` parameter is the type of the state and result of a fold that uses such algebra.

The **regular algebra** is meant for either manipulating the three or performing some effectful computation while traversing the tree. The result or state of of folds that use that algebra are of the same type as the structure they traverse (ie. they are of type `alg m s s`). The most common use of the algebra is performing applications or substitutions. The function `baseAppAlg` defines an algebra that performs application to a lambda calculus like term.

The **group algebra** is meant to collect results from a catamorphism. The `Group` typeclass as defined in this component is equivalent to a monoid. It defines an operation to join to elements of the group and an empty element. The most common use of this algebra is with maps. Since the index of each component dosen't change from fold to fold, a traversal can save local results in a map and look them up in subsequent traversals.

The first refinement of `Fold` is the type class `WithAbstraction`. This type class models structures that can define scoped variables. Variables in this implementation are represented by integer values. The `increment` method of the class increments every variable by the specified ammount. Details about the importance of the function will come later. It also defines the method `abst` which serves to pattern match abstraction elements of the structure and the method `abstC` that serves to build abstractions of that structure.

The `LambdaCalculus` class is a refinement of the `WithAbstraction` class. In addition to abstractions, structures that belong to this class also have variable occurences and applications. As before, it provides the methods `app`, `appC`, `var`, `varC` to pattern match and construct application and variables. The algebras are extedned in a similar fashion.

The final refinement is the `WithSets` class. This is the class of structures that contain sets. A set is modelled as an empty set and a union that joins elements to increase the size of the set. Additionally, the class can also have a case that contains a set of structures. This is a convenience facility for efficiency and to make comparison of elements easier. When the structure is being traversed by the `foldM` method, it is expected that this method un-packs the set case into a sequence of unions. Algebras over theese structures should make no assumptions on how the un-wrapping is done except that elements of the same set will be connected by unions. The algebra for theese structures does not contain a special case for the set branch since it should get converted to unions and handled as it were an union case.

3.2 Equality, Ordering and Normalization of Structures

The analysis requires certain operations to be defined in strucutres. Since it checks for type equality based on certain equality rules and defines algorithms under the assumption that types will be of a particular structure. The first requirement is some notion of structural equality. This is achieved by:

- Having a uniform naming convention for variables.
- Having an ordering defined for elements and ensure that the unions or sets are always ordered with respect to that ordering.

The uniform nameing convention is achieved by nameing each variable according to the number of abstraction nestings that occur until the abstraction that introduces the variable. The depth of the abstractions that occur in a term are defined as follows:

$$\texttt{depths}\ (\lambda x.t) = \{\lambda x.t\ \rightarrow\ 1\}\ \cup\ \{t'\ \rightarrow\ d+1 \mid (t'\ \rightarrow\ d+1)\ \in \texttt{depths}\ t\}$$
$$\texttt{depths}\ (t_1\ t) = \texttt{depths}\ t_1 \cup \texttt{depths}\ t$$
$$\texttt{depths}\ x = \emptyset$$

Using this notion of depth of an abstraction, it is possible to define a nameing convention where the name of a variable is always the depth of the abstraction where it was introduced. With this notion in place, care must be taken when performing applications since an application changes the depht of all variables inside the terms being applied. To perform applications, first an auxiliary function `increment` is defined (which is the same as the one in the `WithAbstraction` type class):

```
increment i t = increment' t
  where
    increment' (λx.t) = λ(x + i). increment' t
    increment' (t1t) = (increment' t1) (increment' t)
    increment' x when x ∈ freeVariables(t) = x
    increment' x = x + i
```

A basic algebra that performs this increment is is provided in the "Types/Common.hs" called `baseIncAlg`. This algebra implements the common requirements for a lambda calculus and can be extended to handle more complex structures. Now assuming that one has two terms `t1` and `t2` with all variables named according to the depth of the binder where they wehere introduced. The application `t1 t2` is performed by:

1. Incrementing the variables of `t2` by one
2. Perfroming the application (as usual)
3. Decrementing the variables of the result by one

This notion of application allows lambda terms to be named according to the depths and then perform reductions such that terms will reduce to the same terms. It is still the case that terms containing unions have to be ordered in some uniform way to perform equality checks. Fortunately, using the derived `Ord` instance from GHC, the ordering works as needed. This is evidenced by the tests included in the package Analysis.Types which live in the folder test.

To sumarize, the normal form of a term is obtained by:

1. Re-naming the variables according to the naming convention discussed above
2. Reducing the term until a fixpoint is found
3. Grouping all unions that appear in the term inside a Set (from `Data.Set`)

3.3 Reduction

The addition of sets to the language requries a couple of extra reduction rules to ensure normal forms are indeed equal. Theese rules are implemented in "Types.Common" by the algebra `baseRedUnionAlg`. Below they are defined:

`(t1 ∪ t2) t3` → `(t1 t3 ∪ t2 t3)`
`(λ x . t1) ∪ (λ x . t2)` → `λ x . t1 ∪ t2`