

# Experimentation Project

Ernesto Rodriguez

December 1, 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 precision. 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 from the standard input and the output 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 further.

## 3 Types

This module defines all the types that were defined to model the analysis. This amounts 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 operate 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. All maps that indexed by integers used in this implementation use this unique identifier as their 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 tree or performing some effectful computation while traversing the tree. The result or state 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 doesn'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 amount. 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 occurrences and applications. As before, it provides the methods `app`, `appC`, `var`, `varC` to pattern match and construct application and variables. The algebras are extended 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 these 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 these 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 structures. 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 naming convention is achieved by naming 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:

$$\begin{aligned} \text{depths } (\lambda x.t) &= \{\lambda x.t \rightarrow 1\} \cup \{t' \rightarrow d+1 \mid (t' \rightarrow d+1) \in \text{depths } t\} \\ \text{depths } (t_1 t) &= \text{depths } t_1 \cup \text{depths } t \\ \text{depths } x &= \emptyset \end{aligned}$$

Using this notion of depth of an abstraction, it is possible to define a naming 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 depth 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' (\lambda x.t) = \lambda(x+i). increment' t
    increment' (t1 t) = (increment' t1) (increment' t)
    increment' x when x ∈ freeVariables(t) = x
    increment' x = x + i
```

A basic algebra that performs this increment is provided in the `Analysis.Types.Common` package and is 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 were introduced. The application `t1 t2` is performed by:

1. Incrementing the variables of `t2` by one
2. Performing 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 summarize, 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 requires a couple of extra reduction rules to ensure normal forms are indeed unique. These rules are implemented in `Analysis.Types.Common` package by the algebra `baseRedUnionAlg`. Below they are defined:

$$\begin{aligned} (t1 \cup t2) t3 &\rightarrow (t1 t3 \cup t2 t3) \\ (\lambda x . t1) \cup (\lambda x . t2) &\rightarrow \lambda x . t1 \cup t2 \end{aligned}$$

Since the language contains sets, one must ensure that ordering of the elements of the set does not matter. In other words, the elements of a set must always be ordered in a uniform way. To achieve this, the derived `Eq` instance and the structure `Data.Set` are used. For this, the algebra `unions` is defined in `Analysis.Types.Common`. This algebra traverses a structure and combines all elements connected by unions into a single set. This is the reason why the types `Annotation` and `Effect` contain a constructor named `Set`. As mentioned above, the folds for each of the elements re-write these sets as sequences of unions.

The final ingredient of reductions is performing applications as mentioned in the section above.

## 4 Algorithms

The algorithms for the analysis are defined as algebras which are then used to traverse the structure. The four main algorithms are:

1. The completion algorithm (`Analysis.Algorithms.Completion`).
2. The instantiation algorithm (`Analysis.Algorithms.Instantiation`).
3. The join algorithm (`Analysis.Algorithms.Join`).
4. The matching algorithm (`Analysis.Algorithms.Match`).
5. The reconstruction algorithm (`Analysis.Algorithms.Reconstruction`).
6. The constraint solver (`Analysis.Algorithms.Solve`).

Additionally, the package `Analysis.Algorithms.Common` contains code which is common to all the algorithms. The following sections explain details on how these algorithms were implemented.

### 4.1 Common

In this package, mostly structures for bookkeeping the different stages of the algorithm are defined. The structure `RState` is used to store the value of variables created at different stages of the reconstruction algorithm. It contains three fields:

- **freshFlowVars** map that contains the identifiers assigned to the  $\beta_1$  variables of the reconstruction algorithm. This is necessary because the variable is used as argument for a recursive call so all atoms of the `LambdaCalc` need access to its value.
- **completions** map that contains the value that results from calling the completion algorithm in the abstraction branch. The result of the completion is added to the environment and used in recursive calls. However, this map is not entirely necessary since the value could be recovered from the environment but it is defined for convenience.
- **gammas** This contains the value of the environment  $\Gamma$  at every point in the reconstruction algorithm.

The values of the components of this structure are initialized by the functions `calcCompletions` and `calcGammas` of the module `Analysis.Algorithms.Reconstruction`. The `RContext` structure is used as the state of the state monad where the reconstruction algorithm is being executed. It contains the fields:

- **freshIx** this field contains the value of the latest index that was given to a fresh variable. Fresh variables are always given a negative index and every time a new fresh variable is demanded, the value of this field is decreased by one.
- **fvGammas** this field contains a map which stores the sort that has been given to every fresh variable created at any stage of the algorithm
- **history** at every step of the reconstruction algorithm, the value that was assigned to all the variables present in that step is saved in this field. This allows easier inspection on how the final result was obtained.

The reconstruction algorithm also contains an Exception monad to handle cases when an incorrectly typed lambda term is analyzed. To nicely display error messages, the `FailureElement` type is used. Details about this type are very technical because it is simply a choice type with many branches. The type is defined this way so different rendering mechanisms can display errors in the most suitable way.