Experimentation Project

Ernesto Rodriguez

December 2, 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. 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 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 `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 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 unique. Theese rules are implemented in `Analysis.Types.Common` package 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`

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 theese 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 theese algorithms were implemented.

4.1 Common

In this package, mostly structures for bookeeping 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 β_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 Γ at every point in the reconstruction algorithm.

The values of the components of this strucutre are intialized 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 alorithm, 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 reconstruciton 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.

4.2 Reconstruction

The algorithm is implemented very similarly as it is described in the paper. It is divided in three stages, namely `calcGammas`, `calcCompletions` and `reconstructionF` which perform the steps indicated above. It includes a logging and error reporting facility. The log contains a list of values for all the variables that are defined in the pseudo-code algorithm of the original paper. Since this implementation performs a particular normalization, the log also includes the values of the variables before normalizations were applied. After the algorithm completes, a final constraint solving step is invoked to obtain the final annotation for the type and the set of effects that have been obtained for the type.

With the algorithm as described in the original paper, the author did not manage to produce a successful implementation to analyzie fixpoints. Below the problems encountered are described and later the solution the author implement in this version of the algorithm. If the implementation as described in the original paper is desired, the program can be compiled with the CPP flag "-DNoFixWorkaround". This is the version used to show the problems.

The first problem is that free variables are produced as a result of replacing a variable with itself during the matching phase of the fix case. Consider the expression:

$$\left(\left(\textbf{fix}\ \left(\lambda x^1 : \mathtt{B} \to \mathtt{B}\ .\ \left(\lambda x^2 : \mathtt{B}\ .\ \left(\textbf{if}\ x^2\ \textbf{then}\ \left(x^1 * (\textbf{False})^{@8}\right)^{@6}\ \textbf{else}\ (\textbf{True})^{@9}\right)^{@4}\right)^{@3}\right)^{@2}\right)^{@1} * (\textbf{True})^{@10}\right)^{@0}$$

and its resulting type and effects (without the additions:

$$\tau = \mathtt{B}^{\{\beta^{-25} * \{\};\ @9;\ \beta^{-25} * @8\}}$$
$$\phi = \{(@0, @3);\ (@1, @2);\ (@4, @10);\ (@6, @3);\ \delta^{-24} * @8\}$$

When the fixpoint is being analyzied. The first step is a recursive call to the reconstruction algorithm to obtain the type of the underlying lambda. This results in:

$$\forall \beta^1 : \mathtt{A}\ .\ \forall \delta^2 : \mathtt{A} \to \mathtt{E}\ .\ \forall \beta^3 : \mathtt{A} \to \mathtt{A}\ .\ \left(\left(\forall \beta^4 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^4}\right) \xrightarrow{\delta^2 * \beta^4} (\mathtt{B})^{\beta^3 * \beta^4}\right)^{\beta^1}\right)$$
$$\xrightarrow{\{\}} \left(\forall \beta^4 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^4}\right) \xrightarrow{\{(@4, \beta^4);\ (@6, \beta^1);\ \delta^2 * @8\}} (\mathtt{B})^{\{\beta^3 * \{\};\ @9;\ \beta^3 * @8\}}\right)^{@3}$$

Important to note is that it has the β^3 and δ^2 variable quantified. Since this is a function that takes as first argument a function and then uses that function in its body (the application labeled **@6**), the β^3 and δ^2 can be substitued according to the function provided in the first argument. Such substitution must also appear in the resulting value and effects because the function is used inside the body. Now this type is instantiated (via the instantiation algorithm) which simply removes the quantifiers making β^1, δ^2 and β^3 free. The type is then pattern matched to obtain the variables τ' and τ''. Doing as stated above, their value becomes (negative identifiers denote free variables):

$$\tau' = \forall \beta^1 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^1}\right) \xrightarrow{\delta^{-24} * \beta^1} (\mathtt{B})^{\beta^{-25} * \beta^1}$$

$$\tau'' = \forall \beta^1 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^1}\right) \xrightarrow{\{(@4, \beta^1);\ (@6, \beta^{-23});\ \delta^{-24} * @8\}} (\mathtt{B})^{\{\beta^{-25} * \{\};\ @9;\ \beta^{-25} * @8\}}$$

Theese types are then matched to produce a replacement. Matching proceeds by adding a replacement to the quantified and then the case for arrows of the matching algorithm is reached. Looking at τ' and τ'', theese are the values of the relevant variables involved in this stage of matching:

$$\delta_0 \overline{\chi_i} = \delta^{-24} * \beta^1 \qquad \phi = \{(@4, \beta^1);\ (@6, \beta^{-23});\ \delta^{-24} * @8\}$$
$$\beta_0 \overline{\beta_j} = \beta^{-25} * \beta^1 \qquad \psi_2 = \{\beta^{-25} * \{\};\ @9;\ \beta^{-25} * @8\}$$

At this stage, the substitution Ω is extended with $\left[\delta^{-24} \to (\lambda \beta^1 : \mathtt{A}\ .\ \{(@4, \beta^1);\ (@6, \beta^{-23});\ \delta^{-24} * @8\})\right]$ and $\left[\beta^{-25} \to (\lambda \beta^1 : \mathtt{A}\ .\ \{\beta^{-25} * \{\};\ @9;\ \beta^{-25} * @8\})\right]$. Both of theese cases substitute a free variable with an expression that contains the variable. One simple solution seems to be eliminating the exprssions that contain δ^{-24} and β^{-25}. For the annotation signature that would give the correct result but for the effect signature, the consumption of **@8** by 'some' expression which clearly happens in the expression labeled **@6**. The second alternative one could consider is quantifying over thoose variables which would result with the fixpoint expression:

$$\textbf{fix}\ \left(\lambda x^1 : \mathtt{B} \to \mathtt{B}\ .\ \left(\lambda x^2 : \mathtt{B}\ .\ \left(\textbf{if}\ x^2\ \textbf{then}\ \left(x^1 * (\textbf{False})^{@8}\right)^{@6}\ \textbf{else}\ (\textbf{True})^{@9}\right)^{@4}\right)^{@3}\right)^{@2}$$

to have type:

$$\forall \delta^{24} : \mathtt{A} \to \mathtt{E}. \forall \beta^{25} : \mathtt{A} \to \mathtt{A}. \forall \beta^{1} : \mathtt{A} \; . \; \left((\mathtt{B})^{\beta^1} \right) \xrightarrow{\{(\mathbf{@4}, \beta^1);\, (\mathbf{@6}, \mathbf{@3});\, \delta^{24} * \mathbf{@8}\}} (\mathtt{B})^{\{\beta^{25} * \{\};\, \mathbf{@9};\, \beta^{25} * \mathbf{@8}\}}$$

The type is equally bad because it introduces poisoning due to the fact that the label **@8** is applied to an arbitrary variable of type $\mathtt{B} \to \mathtt{B}$. If one chooses to replace β^{24} with the identity function, the label **@8** will appear as a possible annotation that flows out of the function but the term constructed at **@8** will never flow out of this expression.

To solve the problem, consider for a moment what β^{-25} and δ^{-24} mean. They originally represented the action of an arbitrary function, but in this context the function is no longer arbitrary. The function is now the argument of `fix`:

$$\lambda x^1 : \mathtt{B} \to \mathtt{B} \; . \; \left(\lambda x^2 : \mathtt{B} \; . \; \left(\mathbf{if}\ x^2\ \mathbf{then}\ \left(x^1 * (\mathbf{False})^{@8} \right)^{@6}\ \mathbf{else}\ (\mathbf{True})^{@9} \right)^{@4} \right)^{@3}$$

which has the annotated type given previously. Of particular interest is the type of the output of the function, namely:

$$\forall \beta^{4} : \mathtt{A} \; . \; \left((\mathtt{B})^{\beta^4} \right) \xrightarrow{\{(\mathbf{@4}, \beta^4);\, (\mathbf{@6}, \beta^1);\, \delta^{2} * \mathbf{@8}\}} (\mathtt{B})^{\{\beta^{3} * \{\};\, \mathbf{@9};\, \beta^{3} * \mathbf{@8}\}}$$

It is important to note that β^4 is the variable that denotes the argument of the recursive call of the function. In this example: $\mathbf{False}^{@8}$. The δ^2 variable denotes the effects induced by the recursive function (ie. the function itself). With this notion it is very easy to determine what β^3 and δ^2 (β^{-25} and δ^{-24} respectively) should be:

- If the annotation signature contains the variable β^4 on it's own, it means that the recursive argument may be returend by the function which means that β^{-25} should be the identity function.
- If the annotation signature does not contain the variable β^4 on it's own, it means that the recursive argument is never returned by the function so β^{-25} should be the constant $\emptyset$ function.
- If the effect signature contains a flow parametrized by β^4 (for example $(\mathbf{@4}, \beta^4)$) it means that a recursive call is performed with the element labeled by the flow label as argument. The variable δ^{-24} should be a function that takes as argumetn an annotation and returns all the flows containing β^4. In this example: $\delta^{-24} = \lambda \beta : \mathtt{A}. (\mathbf{@4}, \beta)$.

If the recursive call takes more than one argument, theese function should be adjusted respectively to discard the annotation variables that don't appear in the corresponding annotation and effect signatures of the recursive functions. With theese adjustments in place, the fixpoint expression now has type:

$$\forall \beta^{1} : \mathtt{A} \; . \; \left((\mathtt{B})^{\beta^1} \right) \xrightarrow{\{(\mathbf{@4}, \mathbf{@8});\, (\mathbf{@6}, \mathbf{@3});\, (\mathbf{@4}, \beta^1)\}} (\mathtt{B})^{\mathbf{@9}}$$

and the whole expression results in:

$$\tau = (\mathtt{B})^{\mathbf{@9}}$$
$$\phi = \{(\mathbf{@1}, \mathbf{@2})\,;\ (\mathbf{@4}, \mathbf{@10})\,;\ (\mathbf{@4}, \mathbf{@8})\,;\ (\mathbf{@6}, \mathbf{@3})\,;\ (\mathbf{@0}, \mathbf{@3})\}$$