Experimentation Project

Ernesto Rodriguez

December 6, 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" introduced at [1]. 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 [2]. This implementation can also be run with the Glasgow Haskell Compiler (GHC) via the interactive shell. It also includes a test suite that can be run with GHC. A compiled version of the implementation is available at `http://netowork.herokuapp.com/stuff/polyvariant/`.

2 Practical Information

This section contains some pointers that will make it easier for the reader to review the source code, web interface and the present document. It is recommended that the reader is familiar with the paper [1] before reading through this document.

2.1 Notation Conventions

Notation conventions are used in this document, several are borrowed from the paper defining the analysis [1]. Below they are described:

- β_i will denote variables for annotation expressions in definitions, pseudo-code or references to [1].
- δ_i will denote variable for effect expressions in definitions, pseudo-code or references to [1].
- ϕ will denote an arbitrary effect expression (as in [1]).
- ψ will denote an arbitrary annotation expression (as in [1]).
- ξ will denote an arbitrary annotation or effect expression (as in [1]).
- $\overline{x_i}$ will denote a sequence of x
- β^i will denote variables for annotations in expressions used as examples. When $i > 0$ it will refer to bound variables (even if the binder is not visible because only a sub-expression of a larger expression is shown) and when $i < 0$ it will refer to free variables.
- δ^i will denote variables for effects in expressions used as examples. When $i > 0$ it will refer to bound variables (even if the binder is not visible because only a sub-expression of a larger expression is shown) and when $i < 0$ it will refer to free variables.
- $t * t_1$ denotes the application of t to the argument t_1 in examples. Space will never be an application in the example expressions (but it is in pseudo-code).

2.2 File Structure

The code is divided in three modules: `Algorithms`, `Types` and `Web`. The three modules are inside the module `Analysis`. The `Web` module only contains details about manipulating Dom elements and interfacing with the other two modules so it's not interesting and will not be explained any further.

3 Types Module

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 operates in many of those types.

3.1 Traversals

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 (`foldM`) 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 (`Data.Map`) 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 doesn't change in different invocations of `foldM`, 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 modeled as an empty set (`emptyM` and `emptyC`) and a union (`unionM` and `unionC`) constructor that joins elements to increase the size of the set. Additionally, its members also have a case that contains a set (`setM` and `setC`) constructor. 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 for structures. Since it checks for type equality based on certain equality rules and defines algorithms under the assumption that types will have 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 nested abstractions until the abstraction that introduces it is located. The depth of the abstractions (nesting degree of the abstraction) that occur in a term is defined as follows:

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

Using this notion of depth of an abstraction, it is possible to define a 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' (λx.t) = λ(x + i). increment' t
    increment' (t1t) = (increment' t1) (increment' t)
    increment' x | 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 where 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`). Done by the function `unions` from `Analysis.Types.Common`.

The function that converts a term to its normal form will, from now on, be refereed to as `normalize`.

3.3 Reduction

The addition of sets to the language requires a couple of extra reduction rules (besides application) to ensure equivalent terms have the same normal forms. These 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 collects 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.

3.4 Tests

The `Types` module contains some tests to ensure that the reduction rules operate as required. The tests were heavily used to ensure the normalization of terms works as required by the paper. Nevertheless, it is conjectured that the normalization rules of this implementation do not work in general for a lambda calculus with set unions.

The `CommonTests` sub-module of `Types` implements re-write rules that are derived from the annotation and effect equivalence relations that are defined in the paper. Concretely, the re-write rules that exist in this implementation are `assocEq`, `emptyEq`, `identEq` and `randomReplace` defined as follows:

$\texttt{assocEq} := \xi_1 \cup \xi_2 \rightarrow \xi_2 \cup \xi_1$
$\texttt{emptyEq} := \xi \rightarrow \xi \cup \emptyset$
$\texttt{identEq} := \xi \rightarrow \xi \cup \xi$
$\texttt{randomReplace} := \xi \rightarrow \texttt{maybeReplace}\ \xi$

where,

$\xi_1\ \in_\xi\ \xi_1 = \texttt{True}$
$\xi_1\ \in_\xi\ \lambda\chi : s\ .\xi_2 = \xi_1\ \in_\xi\ \xi_2$
$\xi_1\ \in_\xi\ (\xi_2\ \xi_3) = (\xi_1\ \in_\xi\ \xi_2) \vee (\xi_1\ \in_\xi\ \xi_3)$
$\xi_1\ \in_\xi\ (\xi_2\ \cup\ \xi_3) = (\xi_1\ \in_\xi\ \xi_2) \vee (\xi_1\ \in_\xi\ \xi_3)$
$\psi_1\ \in_\xi\ (l, \psi_2) = \psi_1\ \in_\xi\ \psi_2$
$\xi_1\ \in_\xi\ \xi_2 = \texttt{False}$

$\texttt{maybeReplace}\ \xi_1 \mid \texttt{fresh}(\chi) \wedge \exists\,(\xi_2 : s)\ .\ (\texttt{freeVariables}(\xi_2) \cap \texttt{boundVariables}(\xi_1)) \equiv\ \emptyset\ \wedge\ \xi_2\ \in_\xi\ \xi_1 = (\lambda\ (\chi : s)\ .\ [\xi_2 \rightarrow \chi]\,\xi_1)\ \xi_2$
$\texttt{maybeReplace}\ \xi_1 = \xi_1$

The `assocEq`, `emptyEq` and `identEq` are trivial to understand. The `randomReplace` rule, essentially picks a random term ξ_2 that occurrs inside another term ξ_1, replaces the term ξ_2 in ξ_1 with a fresh variable χ of the same sort and wraps the result in an abstraction of χ applied to ξ_2. It is necessary to check that ξ_2 does not contain variables bound by abstractions of ξ_1. The implementation of this function performs random selection of the term ξ_2.

The testing approach used here is `QuickCheck`. Random annotations and effects are constructed (via the implemented `Arbitrary` instance) and then the rules defined above are randomly applied to one or several of the elements contained in the randomly generated term. Then the new term and the randomly re-written term are normalized and after normalizing they should be equal up to the equality instance `Eq` generated by GHC. This property is expressed by the functions `normalizeEquivalent` defined in the sub-modules `AnnotatonTests` and `EffectTests` of `Types`.

4 Algorithms

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

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 of the 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 analyze fixpoints. Below, the problems encountered are described and later the solutions are presented. 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:

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

and its resulting type and effects (without the additions):

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

The type contains free variables that the constraint solver is unable to resolve. When the fixpoint is being analyzed, the first step is a recursive call to the reconstruction algorithm to obtain the type of the underlying lambda. This results in:

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

Important to note: it quantifies over the β^3 and δ^2 variables. 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 instantiated according to the function provided as argument. Such instantiation must also appear in the resulting annotations and effects because the function is used inside the body. Next the 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):

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

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

These types are then matched to produce a replacement. Matching proceeds by creating a replacement to the quantified variable (β^1) and then the case for arrows of the matching algorithm is reached. Looking at τ' and τ'', these are the values of the relevant variables involved in this stage of matching:

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

At this stage, the substitution θ is extended with $\left[\delta^{-24} \to \left(\lambda\beta^1 : \mathtt{A}\ .\ \left\{\left(\mathbf{@4}, \beta^1\right);\ \left(\mathbf{@6}, \beta^{-23}\right);\ \delta^{-24} * \mathbf{@8}\right\}\right)\right]$ and $\left[\beta^{-25} \to \left(\lambda\beta^1 : \mathtt{A}\ .\ \left\{\beta^{-25} * \{\};\ \mathbf{@9};\ \beta^{-25} * \mathbf{@8}\right\}\right)\right]$. Both of these cases substitute a free variable with an expression that contains the variable. This is how the free variables end up in the resulting type. One simple solution seems to be eliminating the expressions 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**, would get discarded. A second alternative is quantifying over the free variables which would result with the fixpoint expression:

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

to have type:

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

The type is equally bad because it introduces poisoning due to the fact that the label **@8** is applied to an arbitrary function of sort $\mathtt{A} \to \mathtt{A}$. If one chooses to replace β^{25} 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 function.

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:

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

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 its own, it means that the recursive argument may be returned 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 its 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 $\left(@\mathbf{4}, \beta^4\right)$) 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 argument 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, these 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 these adjustments in place, the fixpoint expression now has type:

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

and the whole expression results in:

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

The example above provides the intuition behind the problem. The general solution to the problem is creating a list of adjustment functions since occasions when fixpoint recursion is performed over functions that take more than one argument create more free variables. These adjustments are obtained via the functions `getQuantifiedVars` and `mkFixReplacements`. Below they are described:

$\mathtt{getQuantifiedVars}\ \left(\forall\,(\beta : s)\ .\ \tau \xrightarrow{\phi} \tau'\psi\right) = \left((\beta : s), \phi, \psi\right) \uplus \mathtt{getQuantifiedVars}\ \tau'$
$\mathtt{getQuantifiedVars}\ \tau = \emptyset$

The algorithm is defined for lists ($\uplus$ denotes list append and $\emptyset$ empty list) since the order in which the quantifiers appears is important. The algorithm simply matches each quantified variable with the effects and annotations immediately exposed by the expression inside the quantifier. Next is the definition of `mkFixReplacements`:

$$
\begin{array}{l}
\texttt{filterExp}\ \overline{\chi_i}\ \psi_1 \cup \psi_2 = \texttt{filterExp}\ \overline{\chi_i}\ \psi_1 \cup \texttt{filterExp}\ \overline{\chi_i}\ \psi_2 \\
\texttt{filterExp}\ \overline{\chi_i}\ \phi_1 \cup \phi_2 = \texttt{filterExp}\ \overline{\chi_i}\ \phi_1 \cup \texttt{filterExp}\ \overline{\chi_i}\ \phi_2 \\
\texttt{filterExp}\ \overline{\chi_i}\ \beta \mid \beta \in \overline{\chi_i} = \{\beta\} \\
\texttt{filterExp}\ \overline{\chi_i}\ \delta \mid \delta \in \overline{\chi_i} = \{\delta\} \\
\texttt{filterExp}\ \overline{\chi_i}\ l = \{l\} \\
\texttt{filterExp}\ \overline{\chi_i}\ (l, \psi) = \{(l, \psi_1) \mid \psi_1 \in \texttt{filterExp}\ \overline{\chi_i}\ \psi\} \\
\texttt{filterExp}\ \psi = \emptyset \\
\texttt{filterExp}\ \phi = \emptyset
\end{array}
$$

$$
\texttt{mkFixReplacement}\ \overline{(\chi_i, \phi_i, \psi_i)} = (\lambda\overline{\chi_i}\ .\ \texttt{filterExp}\ (\texttt{last}\ \overline{\psi_i}),\ \lambda\overline{\chi_i}\ .\ \texttt{filterExp}\ (\texttt{last}\ \overline{\phi_i}))
$$

$$
\begin{array}{l}
\texttt{mkFixReplacements}\ \overline{(\chi_j, \phi_j, \psi_j)} = \\
\quad \texttt{let cata}\ (k,\theta_0,\overline{(\chi_i, \phi_i, \psi_i)})\ (\chi, \phi, \psi) = \\
\qquad \texttt{let}\ (\psi,\ \phi) = \texttt{mkFixReplacement}\ (\overline{(\chi_i, \phi_i, \psi_i)} \uplus (\chi, \phi, \psi)) \\
\qquad \texttt{in}\ (* \to k,\ [\beta_k \to \psi]\,[\delta_k \to \phi]\,\theta_0,\ \overline{(\chi_i, \phi_i, \psi_i)} \uplus (\chi, \phi, \psi)) \\
\quad\ (-,\ -,\ \theta) = \texttt{foldl cata}\ (* \to *, [], \emptyset)\ \overline{(\chi_j, \phi_j, \psi_j)} \\
\quad \texttt{in}\ \theta
\end{array}
$$

where

- $[\beta_k \to \psi]$ denotes a replacement for all free annotation variables with sort of kind k with expression ψ.
- $[\delta_k \to \phi]$ denotes a replacement for all free effect variables with sort of kind k with expression ϕ

The algorithm `mkFixReplacements` takes as input a result from `getQuantifiedVars`. The input is the sequence of quantified variables of the type of an expression that is given to `fix` as argument along with the top-level effects and annotations. These effects and annotations are inspected for occurrences of the quantified variables and constant values (`filterExp` performs such check). Expressions containing those effects and annotations become the body of the abstraction. When many variables appear quantified, multiple abstractions are created, each with a higher kind, since the top-level annotations and expressions may contain several of the quantified variables since they are bound by several quantifiers. As an end result, this algorithm produces a set of replacements for the free annotation and effect variables that emerge from the incomplete replacement returned by the matching algorithm.

This algorithm seems to be more complicated that it should, but all these details have to be in place is one wishes correct results when using `fix` with functions that take more than one argument; especially if such functions contain intermediate expressions instead of simply being a sequence of abstractions and then an expression. Examples 11-14 of this program contain such cases.

The second adjustment that was done to the algorithm concerns dealing with infinite recursion. Consider fixing the identity function:

$$
\left(\mathbf{fix}\ \left(\lambda x^1 : \mathtt{B} \to \mathtt{B}\ .\ x^1\right)^{@1}\right)^{@0}
$$

The analysis as defined in the paper will give the type:

$$
\left(\forall\beta^1 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^1}\right) \xrightarrow{\delta^{-10} * \beta^1} (\mathtt{B})^{\beta^{-11} * \beta^1}\right)^{\beta^{-9}}
$$

The variables β^{-11} and δ^{-10} don't concern us here since they are taken care of by the previous adjustment. However, the variable β^{-9} remains free. This is the analysis's way of saying, I don't know what comes out of this. Effectively, this function is an infinite loop and nothing comes out of it. The reason why this happens is that the only constraints given to the fresh β variable created in the `fix` branch of the reconstruction algorithm are variables created during the instantiation phase. It is straightforward to solve this problem: include the fresh variables created during instantiation among the constraints and constrain them to have the empty element included in them. The empty element is defined per sort as follow:

$$
\begin{array}{l}
\texttt{empty A} = \emptyset \\
\texttt{empty E} = \emptyset \\
\texttt{empty A} \to s = \lambda\beta.\texttt{empty}\ s \\
\texttt{empty E} \to s = \lambda\beta.\texttt{empty}\ s
\end{array}
$$

With the addition of this constraint, the function above now has type:

$$
\left(\forall\beta^1 : \mathtt{A}\ .\ \left((\mathtt{B})^{\beta^1}\right) \xrightarrow{\{\}} (\mathtt{B})^{\beta^1}\right)^{\{\}}
$$

It indicates that the expression will not return an element constructed anywhere. It is debatable what should be the right answer here, but for practical purposes, one can expect that the expression will effectively never deliver an answer and nothing will flow out of it.

4.3 Completion

The completion algorithm is implemented in continuation passing style. The `completion'` function takes as first argument a continuation function which will eventually return the completed type and a list of constant values for the constraint solver. The implementation is done very similarly as described in the original paper except that recursive calls are replaced by continuations. The execution (for the arrow case) proceeds by first performing the recursive call to completion and then applying the result to a continuation that corresponds to the remainder of the body of the completion algorithm (denoted by `cont''` in the implementation). It uses the underlying state monad to generate fresh variables and this has the convenience of keeping track of the sort that the free variables are assigned.

The role of the completion algorithm is to annotate a type without annotations. While doing so a list of variables that should remain constant while solving constraints is generated and the reconstruction algorithm passes these variables to the constraint solver.

4.4 Join

The join algorithm takes care of joining two types into one by matching them on their syntactic structure and generating a type that contains the effects and annotations of both of its arguments. The original pseudo-code is very declarative since it requires that variables bound by quantifiers to be equal up to sorting and alpha-renaming. Fortunately, for normalized types, this happens automatically thus the implementation of the algorithm looks almost exact line by line to the pseudo-code. One of the additions of this implementation is that its first argument is the label where the joining is being performed. This is only used for error reporting.

4.5 Matching

The matching algorithm also benefits from the assumption of having its arguments normalized. This allows an almost line by line translation (mostly for the matching of patterns). Two helper functions are defined, namely `getChis` and `getBetas`, which extract the arguments and function of an application. The result, instead of being a function, as done in the pseudo-code, is a map mapping variable names to arguments. The map is indexed by integers. This is possible since the replacements only involve free variables and each time a free variable is generated, a unique index is given to it. The type of the values of the map is `Either Annotation Effect`. This is because the variable for which the replacement applies is an annotation or an effect. This allows it to be used as a replacement for annotation arguments by simply removing all the effects from the map. The application of replacements are performed by the algebra `baseReplaceAlg` defined in `Analysis.Types.Common`. This algebra takes a map indexed by variable names and replacements for the variable and returns an expression with those replacements applied.

4.6 Instantiation

This algorithm removes the outer most quantifiers of a type and replaces the variables bound by those quantifiers with free fresh variables. Not much can be said about the algorithm since it is a line by line translation of the pseudo-code. Free variables are obtained using the context which keeps track of the sort given to the variables.

4.7 Constraint Solving

The algorithm is implemented as described in the paper. Since `Annotation` and `Effect` are different types in Haskell, they are packed inside an `Either` to distinguish them. In addition to the arguments described in the paper, the algorithm takes as an additional argument: the label of the expression under active analysis for error reporting. The constraint solving phase is done by the `solveIt` algorithm which iterates over a queue inside its own state monad. To check for the subset relation, the following rule is used:

$$\xi_1 \subseteq \xi_2 \;=\; \texttt{normalize}\ \xi_2 \;\equiv\; \texttt{normalize}\ (\xi_1 \cup \xi_2)$$

The pseudo-code also has the luxury of being able to apply substitutions in a single line but the implementation of the solver must distinguish which case of the `Either` type is being substituted and then apply the replacement of free variables as appropriate. This process is handled by the sub-routine `lookupExpr`. Since this algorithm lives in the monadic environment from which free variables are obtained, it also has access to the environment containing the sorts of the free variables. This is important because the initialization stage when all variables are assigned to the empty set, the empty element for the variable is used instead of the empty set. This element is obtained as described above using the sort available in the environment.

5 Conclusions

This is a successful implementation of a flow analysis for a first order language. It shows it is possible to reconstruct the type annotations corresponding to effects and flows of the language, even though such annotations are Rank-2 types. The implementation was tested with many examples, in particular the 14 examples included with the distribution, which contain:

- **Example 6** and **Example 7**: A function with a second order type.
- **Example 10**: An infinite fixpoint.

- **Example 13** and **Example 14**: Fixpoint applied to a function of kind $* \to * \to *$

The original algorithm required some additions (described in section 4.2). The author believes that at least these additions are necessary but is unsure if they are the only extra requirements for the algorithm to be complete. The examples that have been tested with this implementation are among the most complex examples the algorithm has been tested with.

For the implementation to be successful, the author needed to create a method to reduce equivalent terms to the same normal form. This method was described in 3.2 and was tested against randomly generated lambda terms under randomly applied re-write rules based on the equality relations described in [1]. The normalization process seems to be sufficient to reconstruct the type annotations and for the equality relations to hold. However, the author believes it is not sufficient for a lambda calculus with set unions. In particular, the following reduction:

$$\chi\ \xi_1\ ...\ \xi_n\ \cup\ \chi\ \xi'_1\ ...\ \xi'_n\ \to\ \chi\ (\xi_1 \cup \xi'_1)\ ...\ (\xi_n \cup \xi'_n)$$

is probably necessary to ensure normal forms of equivalent terms are equal but doesn't seem necessary for this algorithm since the constraint solver (where normalization is essential) will still terminate with the absence of the rule (ie. after one extra iteration both sides of the relation will be in the sequence of unions) and once the variable χ is replaced by another term, the result of the reduction will be the same. The rule was omitted because (even though it looks trivial) it is not so simple to implement so there is no reason to pay for unnecessary increase in complexity. This also shows that the tests shipped with the implementation are incomplete since they still pass without this reduction rule.

The chosen normalization approach enjoys several advantages when implementing code that operates with normalized expressions. For example, the join, instantiation and matching algorithms require almost no extra complexity (compared to their declarative definition in [1]) because the equality checks follow trivially if terms are normalized as described. The naming the normalization used also makes it difficult to confuse a bound variable with the fresh identifier originally given to it: once it becomes bound, its name will change.

Conveniently, the normalized expressions are very nice for rendering results. Since all variables are named according to their depth it is easy to keep track of what is the purpose of each variable when inspecting the results.

The implementation also includes a web based user interface which makes it very pleasant to inspect results delivered by the algorithm. The interface also includes a log containing the values assigned to all the variables involved at each step of the reconstruction algorithm. The variables corresponding to annotated types are presented both normalized and raw (τ^* is the raw version of τ). The log was essential for the author to determine what was missing for fixpoints to be correctly typed. The log is accessible even if the algorithm failed (unless parsing of the expression failed). The interface also implements the reduction rules of the lambda calculus (as described in [1]) so the user can double check that the annotations and effects indeed match the result.

The implementation has some drawbacks. In particular, folds have been implemented by hand instead of using an attribute grammar. This happened because 1) it was a good learning exercise for the author in order to understand how attribute grammars work 2) The author likes all his code being native Haskell. Using some extra type class hackery and the view patterns extension, some algorithms could be implemented generically enough to work with all structures but adding labels to the lambda calculus resulted in a lot of code duplication because of this approach. Another drawback is that the use of continuation passing style made the instantiation algorithm much more complex than necessary (another experiment by the author). To parse expressions, the web interface uses the derived `Read` typeclass which provides bad error messages.

References

[1] Stefan Holdermans and Jurriaan Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 63–74, New York, NY, USA, 2010. ACM.

[2] GHCJS Team. Ghcjs. `https://github.com/ghcjs/ghcjs`.