FunFlow: Control-Flow and Units of Measure Analysis for a Simple Functional Programming Language

Wout Elsinghorst Pepijn Kokke

July 12, 2013

Description

For this assignment we've implemented the following two analyses for a basic functional programming language:

- Control Flow Analysis
- Units of Measure Analysis

For Control Flow Analysis (CFA) we have extended the basic syntax of the FUN language to support the construction en destruction of binary Sums and Products. These kind of types can be named by the programmer and for CFA their creation points will be tracked.

The Units of Measure Analysis (UMA) exposes a few new builtin functions to the programmer to allow him to instantiate integer terms with certain measurement unit type annotations. These annotations are propagated and combined during type inference to aid the programmer in writing unit-correct programs.

The implementation of these analyses follows a two stage approach. In the first stage types are inferred and constraints are generated while in the second stage the constraints are solved. The type inference is done by an our own implementation of algorithm W.

The UMA has some none-trivial constraint solving code. The commutativity of unit multiplication makes it impossible to just linearly follow the constraints. Our implementation interleaves straightforward annotation unification with annotation rewriting. The rewriting aims put the annotations into a normal form that can then be fully unified. Unfortunately, this rewriting is not fully complete and it will probably leave some of the more advanced annotations stuck. Luckily, it's not completely trivial to trigger the generation of unsolvable constraints, and even then, the unsolved constraints are usually descriptive enough to allow the programmer to manually judge the unit correctness of his or her program.

Program Input / Output

The file `src/FUN.hs` contains the program on which the analysis is run. The exact form of this program can be adjusted by tweaking the case statement in `example`. One can choose between a program demonstrating the use of units of measurement and between a program showing various language constructs, which can be used to check flow analysis.

The output of the program is a typed and annotated version of the input program. The convention is that annotation variables are put between curly braces `{ }` while concrete annotations are put between square brackets `[ ]`.

Code Layout / Points of Interest

In `src/FUN.hs`

```
main ::  IO ()
```

Loads the example code. Prints the results.

```
example ::  Program
```

A switch statement coming with a collection of pre-written programs to test the code. Use option 1 to see measurement analysis in action.

In `src/Base.hs`

Basic definitions of the AST and some helper functions.

In `src/Parsing.hs`

Code to parse a basic functional programming language.

In `src/Labeling.hs`

Code to attach unique labels to a specific component. Mostly used for generating the unique Program Points used in Control Flow Analysis.

In `src/Analysis.hs`

```
type Analysis a = ErrorT TypeError (SupplyT FVar (Supply TVar)) a
```

Our W algorothm lives in this Monad. ErrorT is used to report error messages during unification and the two Supplies are used to have fresh streams of both type and annotation variables.

`analyse :: Expr -> Env -> Analysis (Type, Env, Set Constraint)`

Runs W on a given expression and generate the necessary constraints for the respective annotation analyses.

`analyseProgram :: [Decl] -> Either TypeError (Env, Prog, Set Constraint)`

Runs W on a bunch of top level declarations and finalize the Supply monads. Every Decl has the inferred type of the Decls above it available to it via the environment. Type checking happens in a single pass, so Decls don't have access to the types of Decls defined below them.

`unify :: Type -> Type -> Analysis (Env, Set Constraint)`

The function `unify` tries to create a unification for its two arguments. It returns an substituion such that applying the substitution on both arguments makes them have equal types in the underlying type system, but their type annotations are not unified yet. Instead, equality constraints are generated which will be used by a specific constraint solver in a later phase to unify the annotations in a proper way. If type unification is impossible, an error is raised. Is used only by src/Analyses.hs:analysis

`prelude :: Analysis (Env, [Decl])`

Builds an initial environment containing the builtin functions used to give unit annotations to FUN programs. Think 'asKelvin', 'asMeter', etc...

In `src/Analyses/Flow.hs`

`solveFlowConstraints :: Set FlowConstraint -> (FSubst, Set FlowConstraint)`

Takes a set of FlowConstraints and builds a substitution mapping each flow variable to the set program points reaching this variable.

In `src/Analyses/Measure.hs`

`solveScaleConstraints :: Set ScaleConstraint -> (SSubst, Set ScaleConstraint)`

From a set of ScaleConstraints, build a scale substitution that unifies all matching annotations. The resulting substitution is then applied to the resulting type environment to obtain a final annotated type environment.

`printScaleInformation :: Set ScaleConstraint -> String`

Print the set of Scale Constraints. Usually the constraint set is first simplified using `solveScaleConstraints`.

`solveBaseConstraints :: Set BaseConstraint -> (BSubst, Set BaseConstraint)`

Similar to `solveScaleConstraints`.

`printBaseInformation :: Set ScaleConstraint -> String`

Similar to `printScaleInformation`.