Types and Semantics

Assignment Type and Effect Systems

May 30, 2016

Our starting point for this assignment is the control-flow analysis that we defined for a functional language in the lectures. We use a slightly modified syntax (in order to correspond to the notation of the book of Nielson, Nielson and Hankin).

The (abstract) syntax is given by

$$\begin{aligned} e ::= \quad & n \mid b \mid x \mid \mathbf{fn}_\pi\ x \Rightarrow e_0 \mid \mathbf{fun}_\pi\ f\ x \Rightarrow e_0 \mid e_1\ e_2 \\ & \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2 \mid e_1\ \mathbf{op}\ e_2 \end{aligned}$$

In this case, **fun** denotes a recursive function definition, where f is the name used for the recursive invocation. Contrary to the book chapter I have distinguished numerical and boolean constants.

For this language, the book and (if you adapt to the different syntax) the slides provide a definition of a control-flow analysis. We use as our starting point the system that has the separate subeffecting rule that avoids poisoning.

The assignment consists of a few steps:

i. Implement this analysis in Haskell, following the slides/book as closely as possible. The trickiest part will probably be how to deal with the non-syntax directed subeffecting rule.

ii. Adapt the analysis to deal with pairs and lists (following Nielson, Nielson and Hankin, mini-project 5.2): redefine e to add

$$e ::= ..\mid \mathbf{Pair}_\pi(e_1, e_2) \mid \mathbf{pcase}\ e_0\ \mathbf{of}\ \mathbf{Pair}(x_1, x_2) \Rightarrow e_1$$

Two things you need to take into account: (1) if we store a function in a pair, retrieve the function from the pair, and apply that function, we want to know where that function could have been defined. And (2), for every pattern match on a pair, we want to know at which program locations that pair could have been constructed (so control-flow now includes data-flow).

iii. To the result of the previous part add syntax for lists:

$$e ::= .. \mid \mathbf{Cons}_\pi(e_1, e_2) \mid \mathbf{Nil}_\pi \mid \mathbf{lcase}\ e_0\ \mathbf{of}\ \mathbf{Cons}(x_1, x_2) => e1\ \mathbf{or}\ e_2$$

and repeat your treatment.

iv. Change your type system specification to deal with general datatypes:

$$e ::= C_\pi(e_1, \cdots, e_n) \mid \mathbf{case}\ e_0\ \mathbf{of}\ C(x_1, \cdots, x_n) => e_1\ \mathbf{or}\ x => e_2$$

Here $C \in \mathbf{Constr}$ denotes an n-ary data constructor (partial application not allowed!). This type rule need *not* be implemented. The rule should generalize the rules you invented for lists and pairs. NB. the case statement takes care of one constructor at the time, if a given datatype has more than two constructors then nested cases should be used by the programmer.

Note that to do the analysis, you also need to extend the type inferencer to deal with these extensions. I leave it up to you to come up with new types for these constructs.

Deliverables: (1) an implementation of the analysis for the base language with support for pairs and lists, (2) a small set of programs that shows that your implementation behaves as it should (of course, these should include uses of pairs, lists, all other constructs, including a non-trivial example in which functions are stored in and extracted from lists), and (3) a pdf that provides and explains the type rules you used for the parts (ii), (iii) and (iv), possible other changes you made to the other types rules and syntax of types to accommodate your adaptations, and a short description that explains how to compile and run your program.

Bonuspoint: if you want, you can get a bonuspoint by implementing a polyvariant analysis. **Do this only if you have time after finishing the above.** Construct the polyvariant version side-by-side with the monovariant one by first cloning the monovariant version and then modifying it. Do not forget to include a few examples for the polyvariant version that show that indeed polyvariance has been implemented and works. Make clear in your documentation that you have attempted the polyvariant analysis and how to compile and use it.
NB1. If by adding your bonuspoint to your score you obtain more than 10 points, your grade will be 10.
NB2. a base implementation with a parser for the Fun language can be downloaded from the website. The implementation was adapted from a submission made by Pepijn Kokke and Wout Elsinghorst.

Good luck.