Types and Semantics

Assignment Type and Effect Systems

June 7, 2022

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

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

In this case, **fun** denotes a recursive function definition, where f is the name used for the recursive invocation. Contrary to the book chapter we 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 with polymorphism (but without polyvariance) and with the separate subeffecting rule that avoids poisoning.

For the following steps you can receive a maximum grade of 8.5. Code readability is worth 1 point, the other points are as follows:

i. *3.0 pt* 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. *1.5 pt* 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. *2.0 pt* To the result of the previous part add syntax for lists:

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

and repeat your treatment.

iv. *1.0 pt* This task only needs to be performed in the formal specification, not in the implementation: 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!). 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. We leave it up to you to come up with new types for these constructs.

You can get bonuspoints for implememenTing any of these additions:

i. Implement call tracking analysis: extend control flow analysis to also return for an expression, the function which might have been applied in the execution of that expression.

ii. Add polyvariance.

iii. Use subtyping instead of subeffecting. You should provide a set of example programs which benefit from this. Make sure you handle co- and contravariance properly.

iv. Implement a program transformation. Additional bonuspoints for also implementing the ideas from Trees That Grow.

You can get one point for a good implementation of each. If you have other ideas, please discuss them with me to see whether that can count as a bonus. Only start working on bonuses if you have time after finishing the required steps!

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.

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.