A Java Implementation of A Type Inference Algorithm

Yizhou Yu

CS263 Project, May 12, 1997

In this project, I implemented Palsberg's type inference algorithm in Java. The syntax I use is as follows.

$a, b ::= x \mid [l_i : @(x)b_i, i = 1..n] \mid a.l \mid a.l = @(x)b \mid (a) \mid \&(x)b \mid b(a)$ if b is a λ-term

Parenthesized term is used to resolve ambiguity. I also gave method invocation higher priority than method update. My program can accept a method without @-binder(sigma-binder) if the bound variable is never used within the context of the method. This does not affect the typing result. I fully implemented Palsberg's algorithm with subtyping and recursive types. For λ-terms and function applications, I extended his algorithm by using the object encoding which is more restrictive than general λ-terms and users are not allowed to use the internal labels $_arg$ and $_val$.

One occurrence of a subterm has one corresponding node in the parse tree and vice versa. So there is no redundency in the number of nodes in the parse tree. Each node can have one of the four types: bound variable, object, method invocation and method update. A node of type *bound variable* has no subtrees. A node of type *object* has each of its methods as one of its subtrees. A node of type *method invocation* has only one subtree and saves the method label at its own place. A node of type *method update* has two subtrees and saves the method label and the bound variable in the new method at its own palce. This facilitates the conversion from subterms to AC-graph.

I directly construct the AC-graph from the parse tree. So there is no explicit representations for the contraints and automata in my program. The construction is done by traversing the parse tree in post-order and creating

new vertices and edges corresponding to each node. And the type inference class only deals with this graph.

Output or error messages are concatenated into two strings, respectively. In the exception handlers or the main function, these strings are dumped to the text areas in the applet if there is one; otherwise, they are dumped directly to the screen. So I don't need to write different code in every function for each case.

The Java classes and objects are summarized as follows.

- ParseTree: each node corresponds to a subterm.
- Parser: encodes λ-terms and do LL1 parsing.
- TypeInfer: constructs the AC-graph, closes the graph, checks if it is well-formed, and etc.
- UI: entends Applet, sets up some input,output text areas.
- AdjMat: adjacency matrix. An edge insertion operation returns the set of additional edges needed to close the new graph.
- PE: a 2D array of lists of edges. If $x_1 \stackrel{l}{\rightarrow} y_1$ and $x_2 \stackrel{l}{\rightarrow} y_2$, append $[(y_1, y_2), (y_2, y_1)]$ to PE$[x_1, x_2]$.
- Matrix M: keeps track of the accesses to PE.
- A queue of edges Q.

The sequential execution of the program is summarized as follows.

- Scan the input string, if there are some λ-terms, do the encoding.
- Use recursive functions to do LL1 parsing. Each recursive function is responsible for one kind of subterms. Construct a parse tree bottom-up during parsing.
- Construct the AC-graph. Initialize Q to be list of $\leq$ edges in the input graph plus those reflexive edges. Compute PE. Initialize adjacency matrix to be the empty graph. Initialize M to be the 0-matrix.

- Iterative step repeated until Q is empty: Let (x,y) be an edge in Q. Remove it and insert it into the adjacency matrix. Let R be the list which is returned by the insert operation. For (x,y) in R, find the immediate sucessors z of x in the adjacency matrix, and if M[y,z]=0, then set M[y,z]=1 and set M[z,y]=1, and append PE[y,z] to Q.

- Check if the resulting graph is well-formed.

- If the graph is well-formed, output "typable"; else output "not typable".