Fabien Bessez

COMP 321: Design of Programming Languages

Homework 4: Typing

Problem 1. Write a validity inference rule for deriving that a sequence of statements that begins with an initialization is valid. In other words, write a rule for inferring Γ ` T x=e;s1;. . .;sn−1 valid(τ) where T is any type, x is any identifier, and e any expression.

$$\frac{\Gamma \vdash Tx \text{ valid} \qquad \Gamma \vdash e : T \qquad \Gamma, x : T \vdash s_1;s_2;\ldots s_{n-1} \text{ valid}}{\Gamma \vdash Tx = e;\ s_1;s_2;\ldots s_{n-1} \text{ valid}}$$

Problem 2. Build a derivation of ` int x ; x = x + 1; valid(τ) This is a derivation from the empty environment, which is the environment with no bindings. You must also write out an appropriate typing rule for assignment expressions (remember that the value of the expression x = e is the value of e; use this to figure out an appropriate typing rule).

Rule for assignment expressions is:

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \vdash x = e : T}$$

Derivation of int x; x = x + 1; valid is:

$$\frac{\dfrac{\dfrac{\dfrac{}{x:\text{int} \vdash x:\text{int}} \qquad \dfrac{}{x:\text{int} \vdash 1:\text{int}}}{x:\text{int} \vdash x + 1:\text{int}}}{x:\text{int} \vdash x = x + 1:\text{int}} \qquad \dfrac{x \text{ is not in } \_}{\_ \vdash \text{int } x;\ \text{valid}}}{x:\text{int} \vdash x = x + 1;\ \text{valid}}}{\_ \vdash \text{int } x\ ;\ x = x + 1;\ \text{valid}}$$

Problem 3. In class we have discussed type-checking for a language in which the only top-level statements are function definitions. How does type-checking change if we get closer to the language in the lexing/parsing assignment and allow variable initializations (but no variable declarations or function prototypes)? That is, a program is a sequence of function definitions and variable initializations. Using words and formalism, describe the type system. Be sure that top-level variable initializations are treated in a way that reflects our typical intuitions about global

variables: they can be used (in a type-correct way) in any function definition that follows the initialization; they are shadowed by parameters and declaration and initialization statements in function bodies; and they cannot be re-initialized at the top level.

Base Case:
Γ |- [] valid

List of Definitions and Global Variables Case:
Γ Γ. |- f: (T_0 * T_1*…T_{n-1}) ->, x_0: T_0…x_{n-1}: T_{n-1} |- ss valid
Γ Γ. |- f: (T_0 * T_1*…T_{n-1}) -> T |- ds valid

Γ |- d :: ds valid

If we define a program as a sequence of function definitions and variable initializations, the type checking rules are different than a program that just consists of a sequence of function definitions. Global variables in this new definition of a program are tricky because an identifier, x, might be initialized as a global variable and as a function parameter. With our current definition of a program (sequence of function definitions) we cannot distinguish the global variable, x, from the function parameter, x, and consequently the function will not type-check. In order to solve this dilemma, the textbook suggests that we place our environment in a stack. In other words, a block statements environment will be placed on top of the stack and used accordingly. This is seen as the appearance of a dot after Γ. Now whenever the context changes, within the block statement, the environment on the top of the stack changes accordingly and not the stack beneath it. To break it down even further, global variables will be set in the lowest stack position and local variables will be set in the appropriate stack position (not the lowest). With this addition, a program that consists of function definitions and variable initializations should type check.