Type Checking Documentation

Austin Kao, Anurag Kashyap, Mia de Leon

March 10, 2021

General Description

Our type checker, written in SML, for a compiler of the Tiger programming language handles mutually recursive functions and types, including records. It uses the abstract syntax tree that we produce and checks the tree for any type mismatches, throwing errors where mismatches occur.

Expressions

For basic operations, addition, subtraction, multiplication, and division we check both sides of the expression for integers.
For comparisons, like greater than and not equal to, we check both sides of the expressions for types that can be compared, such as two strings.
For assign expressions, we check that the variable on the left exists in the type environment and that the variable type matches the type of the expression on the right.
For an if-then-else statement, we make sure that both the then and else statements return the same type and that the if test returns an int.
For a break expression, we make sure that the expression is inside a loop.
For for and while loops, we check the conditions for the loop and that the body of the loops return units.
For array expressions, we check that the expression matches a specific array type in the type environment. This includes the type that the array consists of.
For record expressions, we check that the expression matches a specific record type in the type environment. This includes all of the fields inside the record.
For call expressions, we look up the function in the variable environment and type check the arguments for the function.
For let expressions, we add any declared types or variables to the type and variable environments, respectively. We allow for mutual recursion in these declarations. We then check the expressions contained in the body.

Types and Variables

For any given variable, we check that the variable exists in the variable environment and return the type associated with the variable. If the variable has a field, we check that the variable has a record and that the named field exists inside that record. If the variable is followed by brackets, we check that the variable is an array.
For any given type, we check that the type exists in the type environment and return the type. If the type is an array, we specify that the type is an array of some type. If the type is a record, we specify that the type is a record with a function that returns the desired fields and their types.

Functions and Type, Variable Declarations

Before we process a group of declarations, we check to make sure that no cycles exist (e.g. type a = b, type b = a) and that no type or function name is declared twice.
For variable declarations, we make sure that the types contained in the declaration either exist in the type environment or are created in the same group of declarations. If there is a variable type constraint, we check that the initial type of the variable matches the constraint type.
For function declarations, we check that each function has some declared return type.
For type declarations, as long as no cycles exist in the group and no type name is declared twice, we can process the declarations normally.

Extra Credit

Instead of using Appel's way for evaluating record types, we defined the record type as a pair of a function and a unique reference. The first argument, the function, takes a unit and returns a (Symbol * ty) list. If any of the fields in the record is the type of the record itself then the corresponding type is again a Types.RECORD initialized with the same function.