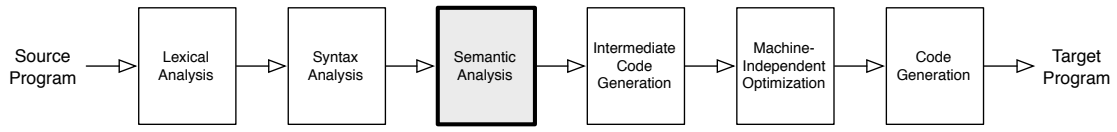
CS 301 Spring 2019
Tutorial Assignment
5 March

Scope, Types

1 Plan



This week, we explore symbol table management for the ROOST compiler and type systems as a foundation for describing and reasoning about type checking. The reading focuses mainly on type systems. Review last week's reading as needed for symbol tables and scope.

2 Readings

- EC 5.5 (review from last week for symbol tables and scope)
- *Type Systems.* **Sections 1–3 (stop after Table 9), Section 9.**
  Luca Cardelli. In *Handbook of Computer Science and Engineering*, CRC Press, 1997.
  `http://lucacardelli.name/Papers/TypeSystems.pdf`

  Even though you will not need to use it in any detail, this paper does assume familiarity with the λ-calculus as a basic syntax for a programming language. The Wikipedia entry for lambda calculus or the first few sections of `http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf` should suffice if you have not taken CS 251.

  Cardelli designed and implemented the first ML compiler and helped to develop much of the formal theory of object-oriented programming. These days, he is working on programming cells and chemicals.
- EC 4.2
- Dragon, Ch 6.3–6.3.2, 6.5. **(Skim)**
- ROOST Language Specification, section on the Core Language Type System

3 Exercises

1. This question applies the symbol table discussion from last week's to write the design of the `scope` package for your ROOST compiler. This package will contain definitions of `SymbolTable` data structure to encode all symbol information for a scope and a symbol table building function that will traverse the AST recursively, constructing symbol tables and annotating each AST node with the symbol table corresponding to its scope, as described in the Front End project document.

   Your design should address the following items:

   (a) Define the general interface to the `SymbolTable`. Describe the operations it supports and sketch the internal representation details. Include a description of what information is stored with each symbol in the tables (*i.e.*, your `SymbolTable` is really a map from identifier to...?).

   (b) Describe the operations performed by the scope-building function upon entry and exit to each form of scope (*e.g.*, top-level, function, structure, signature, method, let-block).

   (c) How will your scope-building function create a new symbol entry when it encounters a declaration?

   (d) When, where, and how will the scoping rules be enforced?

(e) When, where, and how will names be resolved to their declarations?

(f) In what ways, if any, will you change the AST package to support construction of the symbol tables, checking of scoping rules, and resolution of name references? How can Scala traits make this more straightforward?

(g) For each occurrence of ID in the ROOST grammar, identify whether it is a definition that introduces a new name into the current scope or a reference that uses a name from the current scope. For references, indicate the scope in which the defining occurrence of the referenced name would occur.

2. Sketch the contents of your `SymbolTable` structure after processing the following program, which uses type parameterization (similar to Java or Scala generic types) plus structure methods and signatures (like Java instance methods and interfaces) from the standard extensions to the Roost core language.

```
sig S {
  method n(a: i64) -> i64
}

struct A<T> impl S {
  field x: i64
  method m(y: i64, x: T) -> T {   // (c)
    let x = 1 in {
      y = x + self.x;             // (a)
      self.n(y);
    }
    x                             // (b)
  }
  method n(y: i64) -> i64 { 3 }
}

struct B impl S {                 // (d)
  method n(y: i64) -> i64 {
    if (y > 0) {
      f(self, y - 0);             // (e)
    }
  }
}

fn f(s: S, a: i64) -> i64 {
  if a == 0 {
    0
  } else {
    s.n(y - 1)                    // (f)
  }
}
```

3. Use your symbol table sketch to indicate how your compiler will resolve the following symbols:

(a) The names `y` and `x` (2 separate occurrences) in the line marked (a).

(b) The name `x` in the line marked (b)

(c) Both occurrences of the name `T` in the line marked (c).

(d) The name `S` in the line marked (d).

(e) The names `f` and `self` in the line marked (e).

(f) The name `n` in the line marked (f).

4. Using the type system of Cardelli's paper, prove that

   (a) $\emptyset, x : Nat \vdash succ\ (succ\ x) : Nat$

   (b) $\emptyset, x : Nat \vdash if\ true\ then\ x\ else\ 0 : Nat$

5. For each of the following ROOST constructs, state whether it is well-typed in some well-formed typing context, according to the type system from the ROOST Language Specification. If the construct is well-typed, give the most general typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why. (You do not need to prove that environments and types are well-formed. You should be able to convince yourself that any type or environment that you mention is well-formed, however.)

   (a) `(new i64[x.length])[x[2]]`

   (b) `if (x == v[x] && y == "true") { x = y; }`

   (c) `((a == b) == c) || (a == (b + "c"))`

   (d) `f(x)[x.length] = y[2]`

   (e) `if (x == a[b[x]] && y) { y = b[c[x]]; }`

   (f) `f(y, g(x))[1] = g(f(x.length, null))`

   (g) `if (x[a.length] == a.length) { x } else { a }`

6. Suppose we extend ROOST with tuples of the following form. A tuple type is written as a sequence of types in parentheses. For example, the type `(i64, bool, str)` represents a 3-tuple. The individual elements of the tuple can be accessed (*i.e.*, read or written) in a manner similar to array elements. For example, if `x` has type `(i64, bool, string)`, the expression `x[0]` has type `i64`, `x[1]` has type `bool`, and `x[2]` has type `str`. Tuples are unlike arrays in that the index expression must be an integer literal. For simplicity, we assume that tuples do not contain structure or signature types as element types.

   (a) Explain why it is necessary to require that the index of a type expression be a constant.

   (b) Write additional typing rules in the static semantics of ROOST for expressions and statements to support tuples.

   (c) Consider the types `T1 = (i64, (i64, i64)[])` and `T2 = (i64, (i64, i64))[]`. Consider a variable `x` having either type `T1` or type `T2`. Write an expression that type-checks and has the same type in both cases. Write an expression that type-checks if `x` has type `T1`, but does not type-check if `x` has type `T2`. We require that `x`, `0`, and `1` are the only variables and constants in your expressions.

   (d) Syntactically, the tuple element access expression looks like an array element access expression. Will this create problems for type checking? Explain briefly.

7. This problem concerns the typing and the translation of `for` loop constructs.

   (a) Suppose we extend ROOST to support for loops that operate over ranges of integer values: `for (x from e1 to e2) b`. In this where `x` is an `i64` local variable (that has been declared before), while `e1` and `e2` are arbitrary expressions providing the loop bounds, and `b` is an arbitrary block expression: the body of the loop. The loop body, `b`, is executed for all values of `x` in the range from the result of evaluating `e1` up to but not including the result of evaluating `e2`. Write a typing rule that ensures the safe execution of this loop.

   (b) It is difficult to reason about for loops when the execution of the loop body might change the iteration variable or the loop bounds. Describe a semantic check that would ensure this never happens. (You can either write down typing rules to capture your checking or just explain your checking in English.)

(c) Suppose we further augment ROOST with extended for loops, in a fashion similar to those in Python, Java 1.5, or Scala. An extended for loop in ROOST will have the following form: `for (x: T in e) b` Here, `x` is a newly declared local variable of type `T` that iterates over all of the elements of the array resulting from evaluating the expression `e`; `b` is the loop body, a block expression. Write an appropriate typing rule for this construct.

8. Section 1 of Cardelli's *Type Systems* as well EC 4.2 discuss nominal (name) and structural type equivalence. What is the difference? Which does Java (and ROOST) use? What about other statically-typed languages you have used (*e.g.*, ML, Scala, Haskell, C\, ...)? In dynamically-typed languages (*e.g.*, Racket, Scheme, Python, Ruby, JavaScript, ...?) Do any languages mix the two? Both authors describe tradeoffs between nominal and structural typing. Do you agree with them? Which is better? Which issues should you worry about?