Lab2

Diony Rosa

October 2, 2014

Contents

1 Data Structures

1.1 Symbols

Relevant files: `scala/compile/SymbolTable.scala` Symbol is a base class that represents any kind of object that can be stored in a symbol table. All symbols have an `id: ID` member, which is the name of the symbol.

1.1.1 FieldSymbol

A field symbol represents a variable (a non-method and non-callout). Fields have the following members:

- `dType:DType`: The type of the variable if its scalar, or the type of each element, if the field is an array

- `size:Option[Long]`: If the variable is a scalar, this is None. Otherwise it is the length of the array

1.1.2 CalloutSymbol

Callouts only store their id.

1.1.3 MethodSymbol

This represents a method declaration Methods have the following members:

- `params: SymbolTable`: The symbol table that any nested scopes should use as their parent. This symbol table contains the method's arguments.
- `returns:DType`: The return type of the method.

1.2 SymbolTable

Relevant files: `scala/compile/SymbolTable.scala` `SymbolTable` represent scopes. Symbol tables store two things: a list of all of the variables in their scope, and a parent the parent scope. Symbol tables support the following operations:

- `addSymbol(symbol: Symbol) : Option[Conflict]` Attempts to add a symbol to the table. If another symbol with the same name exists in the current scope, a `Conflict` will be returned. The `Conflict` object keeps track of the first symbol found, and the duplicate second symbol.
- `addSymbol(symbols: List[Symbol]) : List[Conflict]` Calls `addSymbol` on each symbol, returning a list of all the conflicts encountered.
- `lookupSymbol(id: ID) : Option[Symbol]` Attempts to find a symbol by name in the current scope. This method will also attempt to look in its ancestor's scopes if the symbol was not found in the local scope.

1.3 Intermediate Representation

Relevant files: `scala/compile/IR.scala` The semantic checker phase of the compiler converts the parse tree into an IR. The IR has the following types:

- `ProgramIR(symbols: SymbolTable)` Represents a valid DECAF program.

- `Block(stmst: List[Statement], fields: SymbolTable)` A list of statements, and the symbol table associated with that scope

1.3.1 Statement Types

These types represent all of the possible DECAF statements

- `Assignment(left: Store, right: Expr)`
- `MethodCall(method: MethodSymbol, args: List[Either[StrLiteral, Expr]])`
- `CalloutCall(callout: CalloutSymbol, args: List[Either[StrLiteral, Expr]])`
- `If(condition: Expr, thenb: Block, elseb: Option[Block])`
- `For(id: ID, start: Expr, iter: Expr, thenb: Block)`
- `While(condition: Expr, block: Block, max: Option[Long])`
- `Return(expr: Option[Expr])`
- `Break`
- `Continue`

1.3.2 Expression Types

These types represent all of the possible DECAF expressions

- `BinOp(left: Expr, op: String, right: Expr)`
- `UnaryOp(op: String, right: Expr)`
- `Ternary(condition: Expr, left: Expr, right: Expr)`
- Load Types
  Loads are a sub-type of expression.
  - `LoadField(from: FieldSymbol, index: Option[Expr])` Represents loading a value from a variable or array location
  - `LoadLiteral(inner: CommonLiteral)` Represents a constant
  - `Store(to: FieldSymbol, index: Option[Expr])` Represents saving to a variable or array location

2 Semantic Checking

Relevant files: `scala/compile/ASTToIR.scala` The class `IRBuilder` takes an AST as an argument, and walks down the tree. As it traverses the tree, it calls a variety of `convert` methods, such as `convertAssignment`, which convert AST types into their corresponding IR types, while doing semantic checks. Any of these methods can fail, which will cause add an error to a shared error list, and will propogate its failure up the tree.