Project 2 Design Document: Semantic Checker

Maddie Mott (mmott@mit.edu), DeJuan Anderson (dejuan@mit.edu), Youyang Gu (yygu@mit.edu), Desai Chen (desaic@csail.mit.edu)

- Division of labor
    - IR nodes classes: Maddie and Youyang
    - IRMaker: Maddie
    - Descriptor Classes: DeJuan
    - JavaDocs: DeJuan
    - AST parser modifications: Desai
    - Documentation: Desai
    - Additional Tests: Youyang

Overview

Our internal representation of a Decaf program consists of a collection of symbol tables, which contain, among other attributes, trees of IR nodes. We use the IRMaker class to build IR tree from the AST output by our parser while verifying the semantic validity of the program. By having separate classes for representing and building the IR, we can keep the IR node classes simple and easy to use for later stages. The output of the IRMaker class is a global symbol table, mapping from the string names of callouts, fields, and method declarations to their Descriptors. While building the IR, we populate the global symbol table and a stack of local symbol tables on the fly while traversing the AST and building up IR nodes which represent the code of method declarations.

IR Classes

Our IR trees represent each method defined in a Decaf program. Since in a Decaf program, variables must be declared before they can be initialized, we can isolate field declarations into the symbol tables and not include them directly in the IR. The alternative would be to have more IR node types to represent variable declarations and function declarations.

Our classes for IR_Nodes all inherit from the abstract class IR_Node, and frequently have other IR nodes as attributes (e.g., an IR_ArithOp has an IR_Node for its left-hand and right-hand sides). The IRMaker class is the main point of interaction with these nodes. For each kind of node it expects to encounter while walking the AST, it has ValidateX and GenerateX methods. While we could have built the validation into the generate function, isolating the validation means that generate functions can assume they can complete construction safely, at the cost of repeating work done in the validation functions. The public function provided by IRMaker is GenerateProgram, while takes as an argument the AST output by the parser, and returns the global symbol table containing all information we will need in later steps.

IR Node Hierarchy

- IR_Node
  - IR_Literal
    - IR_IntLiteral
    - IR_BoolLiteral
    - IR_StringLiteral
  - IR_Seq - sequence of statements in a block
    - While not descended for IR_Seq, these nodes appear within it:
    - IR_If
    - IR_For
    - IR_Call
    - IR_Return
    - IR_While
    - IR_Continue
    - IR_Break
    - IR_STL, IR_STP, IR_STF, IR_STA
  - IR_Var - an int, bool, int[], or bool[] variable, typically used within a store or load operation
  - IR_ArithOp
    - IR_ArithOp_Mult
    - IR_ArithOp_Sub
    - IR_ArithOp_Plus
    - IR_ArithOp_Div
    - IR_ArithOp_Mod
  - IR_CondOp
    - IR_CondOp_And
    - IR_CondOp_Or
  - IR_CompareOp
    - IR_CompareOp_LT
    - IR_CompareOp_LTE
    - IR_CompareOp_GT
    - IR_CompareOp_GTE
  - IR_EqOp
    - IR_EqOp_Equals
    - IR_EqOp_NotEquals
  - IR_Ternary
  - IR_LDL, IR_LDF, IR_LDP, IR_LDA

Symbol table structure

We maintain a global symbol table and a list of local symbol tables while traversing the AST. The global table has entries for callout declarations, global fields and method declarations, and maps for the string identified to the corresponding descriptor. Each method's IR node, as well as other relevant information, are held in its Descriptor. The lists of local tables are used to keep track of local variables and local array lengths. Whenever we enter/exit a block or a method declaration, we push/pop a local symbol table. The local table maps from a symbol name to the corresponding Type. The global table maps from a symbol name to subclasses of the Descriptor class that contain more information, most importantly function parameters, IR_Nodes, and return type in MethodDescriptor.

We considered alternatively using a single stack of symbol tables, each mapping from symbol names to its IR node. However, since we decided to remove declarations from our IR, we no longer store such information. Separating symbol tables into a global one and a local stack makes it easier for us to understand their contents and to debug.

IRMaker

The IRMaker performs semantic checks and builds an IR from an AST. In our current implementation, if the semantic check fails at any step, the IR building process would stop. This simplification allows us to prototype the semantic check more quickly. We will implement a more generous checker that can report more errors at once.

Alternative: We considered using a visitors pattern to build the IR from an AST and report errors. However, this would lead to more fragmented code and take longer to build. So for now, all of the logic resides in one file. We might revisit this when considering optimization.

Modifications to Previous Stage

We modified the parser and scanner to make ANTLR produce a minimal homogeneous syntax tree instead of a list of tokens. We then build our IR in Java instead of augmenting the parser rules. We chose this approach to get away from the parser generator as early as possible and avoid any trouble caused by ANTLR.

Testing

We added a copious amount of additional tests that attempt to comprehensively cover each of the 23 semantic rules. We have 48 total illegal tests and 8 legal tests. The legal tests consists of 7 which are not only semantically legal, but should also be successfully executed at runtime. These tests will come in handy in later steps. The last legal test tries to fool the IR generator by generating a lot of long and useless (but semantically legal) expressions (including lots of nesting and shadowing). If our semantic checker can distinguish the legal test cases from the illegal test cases, we can be fairly confident about the overall correctness of the checker.

Known issues and explanations

The current implementation of the IRMaker has an inefficiency causing the compile time to increase rapidly with the tree depth of the AST. This is because each node in IRMaker has a generate method, which calls validate on its children. But the validate method can also call generate, creating an potentially exponentially deep tree (e.x. 1+1+1+1+1+1+...). This is the current implementation:

Consider the expression *a1+a2+a3+a4*:
To generate IR, here is the current calling process:

```
validate(a1)
validate(a2+a3+a4)
  validate(a2)
  validate(a3+a4)
    validate(a3)
    validate(a4)
    generate(a3)
    generate(a4)
  generate(a2)
  generate(a3+a4)
    validate(a3)
    validate(a4)
generate(a1)
generate(a2+a3+a4)
  validate(a2)
  validate(a3+a4)
    validate(a3)
    validate(a4)
  generate(a2)
  generate(a3+a4)
    validate(a3)
    validate(a4)
    generate(a3)
    generate(a4)
```

We plan to improve this process to a linear-time solution like the following:

```
generate(a1+a2+a3+a4)
  generate(a1)
    validate(a1)
  generate(a2+a3+a4)
    generate(a2)
      validate(a2)
    generate(a3+a4)
```

```
    generate(a3)
      validate(a3)
    generate(a4)
      validate(a4)
    validate(a3+a4)
  validate(a2+a3+a4)
validate(a1+a2+a3+a4)
```

We currently do not check for variable initialization, array index out of bounds (other than non-negativity), etc. These will be left to be checked at runtime.