Cocopiler

Status

The compiler is currently capable of generating code for most programs. However, there are some programs the compiler cannot currently handle. Most notably the code generator cannot handle arrays. There are also some lingering bugs around how global variables are handled, the are some bugs in the optimizer (most of these lead to suboptimal code but not interfere with code generation), the register allocator does not fully take advantage of opportunities to eliminate register copies, changes made by the optimizer are not highlighted, the code generator cannot handle function overloading, and the code generator is not efficient in its use of memory or the number of operations it generates. See the list of known problems for more details.

Design

This compiler can be thought of as having a series of levels through which the program code follows before ultimately exiting as DLX code. Those levels and their designs are explained below.

1. Scanner

   The scanner reads the original program provided as input and parses it into tokens.

2. AST

   This level may be more accurately called the parser, but it is implemented in the AST class. The AST takes the tokens output by the scanner and builds a structure out of the tokens.

   In my implementation the AST is object orientated, so there is a different class associated with each type of node. This resulted in a massive number of classes which made building the AST easy but resulted in problems later on. Particularly it resulted in a large amount of code duplication. It also turned out to be convenient to turn some nodes into the same type, but that resulted in essentially dead classes which inherited functions they didn't need.

3. Type Checker

   This level scans the AST and makes sure that there are no type errors.

   The type checker is part of the AST and is implemented through a pair of functions: getType and checkType. Calling checkType on a node checks the type of that node and recursively checks the types of all its children. getType is used to support checkType. This allows the programmer to call checkType on the root AST object and have the entire program type checked.

4. IR

   This level takes the AST and simplifies it. It is turned into a series of blocks. These blocks can only be entered at the beginning and exited at the end and are linked to the blocks that enter them and the blocks they exit to. These blocks are filled with instructions that can have an assignee, an operation, and up to two operands. Some instructions may not use all of these, such as a void function call. Function calls always have their own instructions, and the two-operand limit does not forbid functions with more than two parameters.

I implement my IR with three classes: Instruction, Block, and Graph. I also use the classes ValueCode and ir.Variables to help build the IR; and an enum, InstructType, provides the different types of operations.

5. Optimization

   This level takes the IR and attempts to make it better.

   Cocopiler implements five optimizations: dead code elimination, constant folding, constant propagation, common subexpression elimination, and copy propagation.

6. Register Allocation

   This level allocates hardware registers for variables in the IR.

7. Code Generation

   This level turns the IR into DLX code.

Known Problems and Possible Improvements

- The code generator cannot handle arrays.
- Arrays are not passed correctly to the register allocator.
- The code generator cannot handle overloaded functions.
- Register allocator does not take advantage of opportunities to remove copy operations.
- Code generator does not take advantage of copies between variables in the same register.
- Code generated for loading global variables after a function call may overwrite function parameters (if they share a register).
- Temporary variables are not reused.
- Optimizer does not consistently or fully remove empty and inaccessible blocks.
- Optimizer assumes that any function call may have modified all global variables. If this can be tightened it would improve the optimizer.
- Minus sign can be misidentified as a negative sign.