MAlice Compiler

Sarah Tattersall Peter Hamilton

November 20, 2011

1 Introduction

MAlice is an imperative language, inspired by Lewis Carroll's book 'Alice's Adventures in Wonderland'.

The syntax of MAlice is based upon events and dialogue which occur throughout the book and loosely follows English prose. The following document outlines our implementation of MAlice and why we made some of our design decisions.

2 Handling Arithmetic Overflow

In the MAlice code examples, there are some issues with the 32 bit architecture when you handle arithmetic operations involving the MAX and MIN integers for the architecture. These appear to be caught at compile time (using the sample compiler) in order to output error messages.

We have programmed MAlice for a 64 bit environment and so these examples actually run fine, however a similar behaviour is encountered if you use the maximum and minimum numbers for a 64 bit architecture, we kept the way MAlice handles the issues but just extended them to the bounds of a 64 bit architecture.

3 Handling Register Allocation

3.1 Dealing With Register Overflow

This was a particularly difficult issue owing to the fact that pushing and popping registers from the stack at the right time to ensure there were no conflicts and that all variables would be available when needed was not a trivial task.

Therefore the way we dealt with register overflow was that any registers which would overflow were put into memory. We then added these memory addresses to the register map so they could be used in place of normal registers whilst keeping our code for translating intermediate code the same.

We had to be slightly careful since for instructions like mov you cannot move an immediate value into a memory address, you have to move it into a register first. As a result, generating the code for the intermediate nodes also takes this into account.

4 Optimisations

4.1 Removing Un-necessary Includes

To reduce the size of the assembly file where possible, we used some logical statements and flags to decide whether to include certain sections of code. These include the 'extern printf' statement, whether we need the statements for formatting integers and characters and the '.bss' section for variable definitions.

4.2 Temporary Registers and Graph Colouring

To assist with register allocation we used used temporary registers to work out a worst case scenario for how many individual registers we would need. We then used graph colouring to go through the temporary registers and calculate which ones could be replaced by the same real registers (or memory locations in the case of overflow).

This greatly reduced the number needed in the generated assembly code. To perform the calculations we worked out the live ranges for the variables used in the source code and from there worked out when and where they could share registers.

4.3 Smart 'mov' Instructions

When the intermediate code is generated, some instructions result in mov instructions going from a register to itself, to get around this we added a logical statement to our mov node which will return no instruction if it would have otherwise resulted in a pointless mov instruction.

5 Coding Style

5.1 Functions Within Functions

In some instances we placed functions within other functions to make them 'private'. For example in code_generator, we don't really want to be able to call solveDataFlow() by itself, only ever as part of the generate function. As a result, if a parent function depends particularly on several other functions which are unused anywhere else, we nest them.

5.2 Object Orientation

We have put similar code and functionality together in classes where possible and throughout our code we have endeavoured to apply an object oriented programming style.
For example, in generating the code, everything takes place inside a CodeGenerator class which

reduces the variables we have to pass around, making our code cleaner, more readable and more maintainable.

5.2.1 ASTNodes and INodes

For example, in parsing and translating malice programs we have used tree structures built upon different node types. Initially the program is loaded into and ASTNode (Abstract Syntax Tree Node) tree, then this is translates into a generic INode tree (Intermediate Node).