Abstract

This project proposes a new technique for retargetable one pass code generation. Most retargetable schemes require two passes for code generation and the use of a large amount of auxillary storage during code generation. Recently a modification has been proposed which generates fast, one-pass code generators. The scheme imposes rather tight restrictions on the input grammar. We propose a scheme here, which is based on the recursive descent parsing strategy. The tool generates a parser, which takes as input internediate code and parses it in one pass indicating a match at each node. Recursive procedure calls indicate propogation of possible choices choices down the tree and parameters are used to return single choices up the tree. Choices are made using cost considerations. The input X86 processor specifications indicate that the number of procedures is not large. Also our condition on the input grammar are far less restrictive than the earlier schemes.

The contributions of this project are the proposal of a new technique for retargetable one pass code generation and the implementation of the test to check if the imput specification is amenable to one pass code generation. Running this test for Intel X86 machine specification and the generation of code are still under consideration.

Previous Work

Most Retargetable schemes require two passes for code generation and also the use of large amount of auxillary storage during code generation. The two passes required are for labeling and selection. An example of a commanly used tool is BURG (Bottom Up Rewrite Generator). The specifications of the machine architecture is given using bottom up rewrite systems. The grammar is in general ambiguous and is cost augmented to eliminate the ambiguity. The specifications have been used to generate finite state tree pattern matching automata, which functions as code generators when augmented with actions that emit code. An input subject tree (possibly an intermediate code tree) is traversed by the generated tree automation, and each time a mtach of some pattern in the specification is encountered, an action is executed, typically emission of code. To design two walks over the intermediate tree it requires allocating and building an explicit intermediate representation tree.

Recently a modification of BURG, GBURG (Greedy Bottom UP Rewrite Generator) has been proposed, which generates fast one pass code generators. It does greedy pattern matching rules. Typically this means that one does not know the parent operators when processing the left (or right) child and therefore non-optimal choices may be made. GBURG requires that all rules for a given operator have the same non-terminal as the left child. Also none of the non-terminals should be derivable from two distinct let child non-terminals via

chained rules. These rules are somewhat restrictive though they permit the generation of a compact code generator that needs to know only the preceeding non-terminal matched and the current operator to make it's next choice.

The various code generators generators differ in the pattern-matching technology they employ and in when they perform dynamic programming. Pattern matching techniques vary widely in theoritical efficiency.

Our Approach

The scheme proposed in the project is an attempt to generate a one-pass code generator from grammar specifications that are subject to as few restrictions as possible. This scheme will lead to code generators that are small, as they do not require the usage of large tables during code generation time.

We have developed a sub-optimal tree pattern-matching algorithm using a recursive descent parser like strategy. If more than one pattern is matched at a particular node then this ambiguity is resolved taking the cost of productions into consideration. Uisng this a code generator generator has been developed, which has an important advantage over previous systems. The code generator does labeling and reduction in a single parsing pass, this will result in considerably faster code generation.

We set out to generate a one pass, perhaps sub optimal, code generator from the grammar specifications that are subject to as few restrictions as possible. However studies need to be carried out to actually test ouit the quality of the code relative to that generated using optimal techniques.

When we resolved the ambiguity between patterns that match at a particular node we considered the cost of the productions. But we did not take into consideration the matches that were obtained previously or those that might be effected for the parent nodes. It might be interesting to evaluate this trade-off to find out in which case is the code more optimized.

Author : Ananth Inamti Sundararaj

Advisor : Prof. Priti Shankar

Date : June 2000

Department : Department of Computer Science

Degree : Bachelor of Engineering