DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

D C C

Geradores e Otimizadores de Código

por

Mariza Andrade da Silva Bigonha

U F M G

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Geradores e Otimizadores de Código

por

Mariza Andrade da Silva Bigonha

Relatório Técnico RT027/89

Belo Horizonte
1989

SUMÁRIO

Sinopse

Este trabalho contém uma investigação sobre o estado da arte com relação as mais recentes ferramentas criadas para o desenvolvimento de linguagens, ressaltanto uma revisão das técnicas e metodologias de geração e otimização automática de código presentes na literatura. Apresenta também, a organização das diversas etapas envolvidas no processo de desenvolvimento de um compilador e algumas tendências da pesquisa corrente na construção dos mesmos.

1 INTRODUÇÃO

O projeto e implementação de compiladores para linguagens de programação é uma parte essencial do desenvolvimento de sistemas de software. Na década de 1970 foram projetadas várias linguagens de programação, tais como, ADA, C, BLISS, PASCAL e FORTRAN 77 entre outras linguagens mais especializadas. Embora a construção de compiladores tenha se tornado mais fácil devido em grande parte ao novo discernimento teórico e ao aparecimento de ferramentas auxiliares, a implementação de um compilador para uma linguagem de grande porte continua sendo uma tarefa difícil. Este trabalho contém uma investigação sobre o estado da arte com relação as mais recentes ferramentas criadas para o desenvolvimento de linguagens, ressaltanto uma análise das técnicas de geração e otimização automática de código presentes na literatura. Ele é composto de 6 Seções, a primeira Seção é a Introdução. A Seção 2 apresenta a organização das diversas etapas envolvidas no processo de desenvolvimento de um compilador e algumas tendências da pesquisa corrente na construção dos mesmos. A Seção 3 apresenta diversas abordagens de geração de código classificadas segundo Ganapathi et alii [GANAPATHI 82b], com enfâse na geração automática de códigos redirecionáveis. A Seção 4 apresenta cinco projetos relevantes de geradores de compiladores , PQCC - Production Quality Compiler-Compiler, [WULF 80], um Sistema Especialista para geração de código de boa qualidade [HARADVALA 84], ECS - Experimental Compiling Systems, [CARTER 77], ACK - Amsterdam Compiler Kit, [TANENBAUM 83], SIG- Sistema de Suporte à Implementação Automática de Geradores de Código, [AVISSAR 85]. A Seção 5 apresenta uma revisão das técnicas de otimização local de código, suas estratégias e metodologias presentes na literatura.de geração e otimização automática de código presentes na literatura. Apresenta também, as diversas etapas envolvidas no processo de desenvolvimento de um compilador e algumas tendências da pesquisa corrente na construção dos mesmos. A Seção 6 é a Conclusão.

Desde o início de 1960, tem havido por parte dos pesquisadores, grande interesse no desenvolvimento de ferramentas que reduzam o esforço necessário a construção de um bom compilador. Estas ferramentas são frequentemente chamadas de "compiler-compiler" ou "translator writing systems". Entretanto, com algumas raras excessões, até o final da década de 1970, os trabalhos apresentados na literatura com respeito ao desenvolvimento

destas ferramentas, focalizavam apenas as fases de análise léxica e sintática dos compiladores. Eles proviam pouco ou nenhum suporte para a fase de geração de código. Assim, geradores de compiladores quase se tornaram sinônimo de gerador de analisadores sintáticos. Ferramentas típicas deste enfoque são YACC [JOHNSON 75] e SIC [BIGONHA 85], etc.

Mais recentemente, na década de 1980, surgiram na literatura várias tentativas com o objetivo de formalizar o processo de geração de código, tendo em vista, a construção de geradores de compiladores. Dentro deste espírito, encontram-se os trabalhos de Wulf [WULF 80], Leverett et alii [LEVERETT 80], Ganzinger [GANZINGER 82], Tanenbaum [TANENBAUM 83], Avissar [AVISSAR 85], Ganapathi et alii [GANAPATHI 82b] e Kessler [KESSLER 86].

2 MODELO DE UM COMPILADOR

2.1 Introdução

Os sistemas de implementação de compiladores mais antigos tratavam o processo de compilação como uma única fase, a qual denominavam de tradução dirigida por sintaxe. Um gerador de compiladores típico da década de 1960 provia ao projetista de compiladores uma técnica para gerar automaticamente analisadores sintáticos ou uma linguagem para especificar analisadores sintáticos.

Por volta de 1970 surgiram abordagens em que as diversas etapas do processo de compilação era considerada em separado. O processo de tradução do programa fonte para programa objeto passou a ser subdividido em uma sequência de subprocessos chamados fases. Por esta época foram desenvolvidas ferramentas para implementar algumas destas fases automaticamente a partir de especificações de alto nível. Estas ferramentas frequentemente exploravam conhecimento de propósito geral derivado de análises teóricas. Algumas destas ferramentas poderiam ser usadas independentemente da geração de compiladores.

Para apreciar os avanços recentes dos geradores de compiladores torna-se necessário entender o que acontece durante cada fase de compilação. Como a função exata de cada uma delas é mais ou menos arbitrária, é conveniente considerá-las responsáveis pela operação de transformar uma representação do programa fonte em uma representação intermediária.

Esta seção apresenta um modelo de compilador constituído de cinco fases: análise léxica, análise sintática, análise semântica, otimização de código e geração de código.

2.2 Análise Léxica

A primeira fase de um compilador, a análise léxica, considera o programa fonte como uma sequência de caracteres. Sua função é agrupar estes caracteres em símbolos que constituirão as unidades básicas a serem manipuladas pelos analisadores sintáticos. Além disto, esta fase é normalmente responsável pelo reconhecimento de comentários, brancos, "strings", etc.

Expressão regular tem sido uma notação útil na descrição das unidades básicas (tokens) em uma linguagem de programação.Em meados de 1960, começaram a surgir linguagens de propósito especiais para especificação de analisadores léxicos a partir de descrição de expressões regulares. Algoritmos eficientes foram desenvolvidos para construir reconhecedores de autônomos finitos. Exemplos de geradores de analisadores léxicos nesta linha são: LEX - A Lexical Analyser Generator de M.E. Lesk [AHO 86] e AWK de Aho, Kernigham e Weinberger [AHO 79].

2.3 Análise Sintática

A análise sintática é responsável pelo agrupamento das unidades básicas emitidas pelo analisador léxico em unidades sintáticas tais como, comandos e expressões. É sua função determinar se o programa fonte satisfaz as regras de formação da linguagem de programação. Caso o programa não esteja sintaticamente correto, o analisador sintático deve determinar as posições em que ocorreram os erros, fornecer o diagnóstico apropriado, bem como recuperar-se dos mesmos para dar continuidade ao processo de análise sintática. Esta fase produz como saída uma representação do programa fonte em alguma linguagem intermediária, tal como notação polonesa posfixada, quádruplas, tuplas ou estrutura de árvores.

Durante as décadas de 1960 e 1970 as pesquisas se concentraram no desenvolvimento de técnicas de análise sintática. Estas técnicas podiam ser classificadas em dois tipos: um tipo lidava com uma classe de gramática mais ampla, entretanto este tipo de gramática gastava muito tempo na análise; outro tipo lidava com uma classe mais restrita porém mais eficiente. Estas técnicas foram classificadas como métodos "top-down" e "bottom-up". Mais tarde, a técnica de análise sintática "top-down" foi formalizada como gramáticas LL. Knuth, em seu artigo "On the Translation of Languages from Left to Right" generalizou a técnica sobre a análise "bottom-up" formalizando-a como gramáticas LR, a classe mais geral de gramáticas livres do contexto, que poderia ser analisada com um autômato à pilha deterministico. Trabalhos subsequentes apresentados por DeRemer, Aho, Johnson and Ullman e muitos outros se sucederam. Devido a esta generalidade e à facilidade de detectar e reconhecer erros sintáticos, estes métodos passaram a ser utilizados em geradores automáticos de analisadores sintáticos.

Possivelmente, a automatização da análise sintática foi o maior avanço registrado durante

os anos setenta no escopo dos geradores de compiladores. Como consequência destes avanços, pesquisas sobre métodos de recuperação automática de erros tiveram um impulso significante, como pode ser visto pelos trabalhos de Burke e Fisher [BURKE 82]; Graham, Haley e Joy [GRAHAM 79]; Johnson [JOHNSON 78], etc.

2.4 Análise Semântica

A análise semântica, normalmente, se refere a verificação de várias restrições de integridade semântica que ocorrem durante o processo de compilação. Estas restrições podem ser: verificação de tipos, verificação da compatibilidade dos parâmetros em uma chamada de subrotina, etc.

Formalismos baseados em gramáticas de atributos e gramáticas affix [AHO 86] foram propostos para descrição de semântica de linguagens de programação. Mas até meados de 1970, poucos geradores de compiladores usavam estes formalismos para automatizar a análise semântica. A principal restrição era que a implementação automática das gramáticas de atributo e affix são ineficientes porque normalmente requerem o armazenamento de toda a árvore sintática na memória. Todavia, desde o início da década de 1980, tem havido muito interesse nesta área de pesquisa. Até que ponto as gramáticas de atributo e affix podem ser usadas na prática em geradores de compiladores ainda não está bem determinada. Trabalhos recentes, [GANAPATHI 82b], [GANAPATHI 85], [GANAPATHI 88], usam estas gramáticas nas fases de geração e otimização de código para descrever o conjunto de instruções da máquina alvo.

2.5 Otimização de Código

O objetivo desta fase é transformar a representação do programa fonte de uma forma intermediária para uma outra forma intermediária mais compacta e mais eficiente.

A otimização de código também foi uma área de muito interesse durante a década de 1970; algoritmos eficientes foram implementados. Descobriu-se que programas estruturados eram mais fáceis de ser otimizados do que os não estruturados. Técnicas sobre fluxo de dados globais foram estudadas. Entretanto estes desenvolvimentos só começaram a ser efetivamente incorporados à geradores de otimizadores de código no início da década de 1980. Vamos encontrar no trabalho de Wulf, [WULF 80], intitulado PQCC, a descrição de algumas estratégias de otimização de código. A otimização de código será vista com mais detalhes na Seção 5.

2.6 Geração de Código

A fase final de um compilador, a geração de código, transforma um programa descrito em uma linguagem intermediária em um programa em linguagem objeto.

Um dos maiores problemas no projeto de um compilador é a geração de um código objeto de boa qualidade. Algoritmos gerais de geradores de código vem sendo estudados desde a década de 1950. Em 1970 Sethi e Ullman publicaram um clássico artigo no Jornal de ACM, "The Generation of Optimal Code for Arithmetic Expression" que descrevia como gerar um bom código para expressões aritméticas usando uma máquina de dois endereços. Em 1976, Aho e Johnson [AHO 76] propuseram um algoritmo para geração de código baseado em gabaritos (templates) que também produzia bom código para expressões, em máquinas mais gerais.

Entretanto, descobriu-se que a geração de um código de qualidade na presença de subexpressões comuns é inerentemente difícil, mesmo em máquinas simples. Além disto, sequências "ótimas" de código para muitas arquiteturas de máquinas existentes até 1980 não eram tão intuitivas. Assim, em muitos casos de interesse prático, um compilador se apoiava em heurísticas e algoritmos "ad hoc". Isto, consequentemente representava um grande investimento de tempo e esforço para gerar um bom código.

A partir de 1980, os trabalhos em geração de código se voltaram mais para a portabilidade do compilador resultante, Veja Seção 3.1.2.

Finalizando, uma outra área de pesquisa que tem despertado interesse desde o início de 1980 e que pode causar um impacto significante na geração automática de compiladores é a especificação formal de semântica de linguagens de programação.

Os aspectos, tanto sintático como semântico são pontos fundamentais na especificação formal das linguagens de programação.

As gramáticas livres-do-contexo, BNF, têm sido a rigor, a especificação sintática de linguagens de programação desde a publicação do Relatório do Algol 60 [NAUR 63]. Uma vantagem no uso de gramáticas livre do contexto para especificar a sintaxe de linguagens é que a partir dela podemos construir automaticamente analisadores sintáticos eficientes.

A especificação de semântica de linguagens é mais complexa. Vários métodos de especificação de semântica foram propostos e progresso considerável tem sido obtido nesta área. Porém, até hoje não há especificação formal de semântica através da qual compiladores de produção possam ser gerados automaticamente. A partir de 1980, especificações de semântica denotacional de linguagem de programação se popularizaram. O trabalho de Peter D. Mosses [MOSSES 78] intitulado "SIS" permitiu a geração automática de um interpretador a partir da especificação da semântica denotacional de linguagens. Entretanto estudos adicionais ainda são necessários para possibilitarem a tradução da especificação

da semântica denotacional de uma linguagem em um gerador de código eficiente.

3 GERAÇÃO AUTOMÁTICA DE CÓDIGO

3.1 Geradores Redirecionáveis de Código

A proliferação de linguagens de programação e de novas arquiteturas de computadores estimulou a busca de métodos que facilitassem a automatização da fase de geração de código dos compiladores. Progressos na automatização desta fase possibilitou o redirecionamento de geradores de código, [WULF 80], [GRAHAM 80], [GANAPATHI 81a], [CATTELL 78]. Como consequência deste avanço, algumas das técnicas de geração de código, propostas por estes pesquisadores, tornaram-se economicamente viáveis e com isto têm tido uma boa aceitação na prática. Ademais, estas técnicas têm sido um veículo promissor no suporte das experiências com arquiteturas novas, a nível de determinar que impacto uma arquitetura em particular tem em relação ao tamanho do código e tempo de execução em um dado sistema.

Ganapathi e Fischer, em seu relatório técnico [GANAPATHI 81a], classificam a pesquisa em geração de código em três categorias: tratamento formal, abordagem interpretativa e abordagem descritiva. Em artigo mais recente Ganapathi et alii [GANAPATHI 82b] classificam a pesquisa existente em geração redirecionada de código em três categorias: geração de código interpretativa, geração de código baseada em reconhecimento de padrão, e geração de código dirigida por tabela descritas nas Seçoës 3.1.1, 3.1.2 e 3.1.3 respectivamente. A relação entre esta classificação e a anterior é a seguinte: a geração de código interpretativa é mais abrangente que a abordagem interpretativa de [GANAPATHI 81a]. A geração de código baseada em reconhecimento de padrões e a geração dirigida por tabela correspondem a abordagem descritiva de [GANAPATHI 81a].

Neste trabalho será adotada a classificação mais recente, todavia antes de descrevermos cada uma delas será abordado o aspecto formal.

Para Ganapathi e Fischer, [GANAPATHI 81a], a abordagem formal concentrou-se apenas nas soluções para expressões aritméticas. Como exemplo desta abordagem ele inclui os trabalhos de Newcomer [NEWCOMER 75], Aho e Johnson [AHO 76].

Newcomer usa uma pesquisa recursiva baseada em heurística denominada "means-end-analysis" [RICH 83] para gerar padrões de código (code templates) ao invés de instruções de máquina, a partir da árvore sintática e de um conjunto de operadores.

Para alguns problemas, é natural considerar os termos da situação corrente e da situação almejada. Dada uma coleção de fatos especificando o problema, o ponto de partida denomina-se estado corrente e o ponto onde desejamos estar denomina-se estado meta.

Por exemplo, no problema do caixeiro viajante, o estado corrente e o estado meta são definidos pelas localizações físicas. Considere uma abordagem para este problema, na qual procedimentos são selecionados de acordo com sua habilidade de reduzir a diferença observada entre o estado corrente e o estado meta. Esta abordagem é conhecida como "means-ends-analysis".

O trabalho de Newcomer é muito restrito e de pouca importância na prática:

1. Ele trata somente com árvores de expressões aritméticas.
2. Arquiteturas de máquinas reais nem sempre são totalmente representáveis no seu esquema de especificação.
3. Seu algoritmo de geração de código pode falhar ao produzir um "code template" devido a possibilidade de um conjunto inadequado de operadores ou devido a uma limitação da profundidade da pesquisa executada pelo "means-end-analysis". Este limite se torna necessário para prevenir o gerador de código de um possível "loop". Além disto, o algoritmo é exaustivo e portanto muito caro para ser usado em problemas complexos como compiladores de produção. A ordem em que as diferenças são consideradas pode ser crítica e o número de permutações entre elas pode ficar grande demais.

Aho e Johnson [AHO 76] consideram um esquema de geração de um bom código similar a exaustiva "força bruta". Eles usam um algoritmo baseado em programação dinâmica [AHO 86], constituído de três fases para gerar sequências "ótima" de código para árvores de expressões. Na primeira fase, a árvore é percorrida de baixo-para-cima. Para cada vértice "v", todas as traduções possíveis de instruções de máquinas da sub-árvore cuja raíz é "v" são usadas para computar um arranjo de custos Cr[v], onde $1 \leq r \leq$ número total de registradores. Cr é o número mínimo de instruções necessárias para calcular a sub- árvore usando "r" registradores. Todas as permutações na ordem de avaliação são consideradas. Ao terminar esta fase é obtida uma sequência "ótima" de instruções necessárias para computar a sub- árvore cuja raíz é "v" e uma ordem de avaliação "ótima" para a sub- árvore de "v". A segunda fase usa o arranjo de custos e percorre a árvore de cima-para-baixo para marcar os nodos que devem ser calculados em posições de memória devido ao número reduzido de registradores. E a última fase caminha em cada sub-árvore marcada e gera código para avaliar a sub-árvore seguida dos armazenamentos apropriados em posições temporárias da memória. As desvantagens deste modelo são: ele trata somente árvores de expressões aritméticas excluindo sub-expressões comuns e não trata instruções especiais.

3.1.1 Abordagem Interpretativa

Para Cattell, [CATTELL 80] nesta abordagem a construção de um gerador de código baseia-se na descrição da máquina puramente estrutural e comportamental. Trabalhos

nesta linha, incluem Miller [MILLER 71], Snyder [SNYDER 75].

Uma característica desta abordagem é a mistura da descrição da máquina com o algoritmo de geração de código. Tais geradores de código são normalmente escritos pelo projetista e não são formais.

Um esquema de tradução baseado em dois níveis foi sugerido no final de 1950 para auxiliar no projeto de compiladores portáteis. Sua função básica consiste em produzir código para uma máquina virtual e então expandí-lo para um conjunto de instruções de uma máquina real. Nesta abordagem, para implementar "p" linguagens de programação em "m" arquiteturas diferentes somente P+M tradutores são necessários ao invés de M*P, ou seja, são necessários "p" "front-ends" e "m" geradores de código. Tais esquemas usam uma linguagem de especificação de gerador de código especialmente projetada para descrever o processo de geração de código e as instruções da máquina alvo.

A geração de código interpretativa está dividida em duas classes. A primeira classe é constituída de interpretadores que implementam o mapeamento um-a-um ou um-para-muitos das instruções entre a máquina virtual e a máquina real. Estes interpretadores são escritos pelos projetistas de compiladores. Exemplos nesta linha incluem os tradutores P-code de V. Ammann [AMMANN 77] e o "Code Interpreter" UCSD Pascal de K. A. Shillington e G. Ackland (Eds.) [GANAPATHI 82b]. Esta classe requer que muitas das decisões da geração de código sejam feitas pelo "front-end" do compilador.

A segunda classe da abordagem interpretativa é constituída de interpretadores que são baseados num conjunto controlado de esquemas (schemata). A representação intermediária consiste em instruções para uma máquina abstrata. A tradução de sequências de instruções abstratas para instruções de uma máquina real é feita interpretando as mudanças ocorridas nos estados da máquina abstrata e gerando instruções de máquina que causam ações equivalentes na máquina real.

Esta interpretação do estado da máquina abstrata permite ao gerador de código substituir múltiplas instruções abstratas por uma simples instrução de máquina. O gerador de código UGEN que será apresentado no final desta seção exemplifica esta classe.

Abordagem de Miller: Miller [MILLER 71] foi o primeiro pesquisador que tentou isolar as características dependentes de máquina do algoritmo de geração de código. Em seu gerador de código, a linguagem fonte é mapeada em sequências de códigos de dois endereços, as quais são macros escritas em MIML "Machine Independent Macro Language". Estas macros especificam o algoritmo de geração de código.

Para redirecionar um compilador para uma nova máquina na técnica de Miller, as macros Iadd e Fadd devem ser re-escritas, entretanto espera-se que o algoritmo representado pela macro ADD não mude.

Por exemplo,

```
macro ADD x, y
    if type of x = integer and type of y = integer
    then Iadd x, y
    else if type of x = float and type of y = float
         then Fadd x, y
         else error
```

A descrição da instrução de soma na máquina alvo é formada pela especificação das macros Iadd e Fadd em OMML (Object Machine Macro Language). Por exemplo a macro Iadd do exemplo anterior ficaria da seguinte maneira na máquina IBM-360:

```
macro Iadd a, b
      from a in R1, b in R2 emit(AR a, b) result in R1
      from a in R, b in M emit(A a, r) result in R
      from a in M, b in R emit(A b, a) result in R
```

Estados são definidos como configurações das localizações dos operandos. Um estado é permitido se dele for possível emitir código sem mudar os operandos de lugar. Toda macro é associada somente a um conjunto de estados permitidos. É responsabilidade portanto do projetista, especificar as transições entre memória e registradores de tal forma que o gerador de código transfira-se automaticamente para um estado permitido, se necessário. Transferência de um operando da memória para um registrador para implementar uma soma memória para registrador (storage-to- register) é um exemplo de mudança de estado.

O modelo proposto por Miller é muito restritivo porque trata somente de avaliação de expressão e esquemas de endereçamento muito simples; não permite indexação, auto-incremento ou endereçamento indireto.

Abordagem de Snyder: O modelo proposto por Snyder, segundo Ganapathi et alii [GANAPATHI 82b] é uma tentativa de implementação de um compilador portátil para a linguagem C [RITCHIE 79]. Seu compilador usa um esquema de tradução em duas fases, muito similar ao modelo de Miller. Na primeira fase o gerador de código percorre a árvore de expressões e gera instruções de três endereços. Os registradores necessários para estas instruções são definidos pelo programador. A segunda fase então, traduz as instruções de três endereços para código assembler da máquina alvo. Snyder usa macros e rotinas escritas na linguagem C para executar análises de casos (case analises) de sequências de códigos. Sua abordagem ignora por completo a otimização de código objeto.

Embora estes interpretadores escritos pelo projetista sejam um avanço em relação aos métodos "ad hoc" empregados nos compiladores monolíticos, eles possuem algumas limitações como mostrado abaixo. Compiladores monolíticos são compiladores escritos para uma linguagem de programação específica para rodar em uma máquina específica.

1. É difícil prover uma variedade de organizações de máquina em uma máquina virtual, isto devido a diversificação de modos de endereçamento, tipos de dados e instruções da máquina alvo.

2. As linguagens de geração de código são fortemente amarradas a uma linguagem ou máquina específica, portanto não podem ser consideradas totalmente portáteis.

3. A descrição da máquina alvo se mistura com o algoritmo de geração de código, portanto, a descrição não pode ser mudada sem que se mude o algoritmo.

4. O implementador tem uma visão muito local do código a ser gerado, dificultando assim a incorporação de otimizações dependentes de contexto, como por exemplo, o uso de indexação ao invés de uma soma explícita, diferenciação entre valores booleanos que devem ser armazenados (expressões) e valores booleanos que necessitam serem testados (predicados).

O gerador de Código UGEN: O gerador de código UGEN [GANAPATHI 82b], é dirigido por "SCHEMA" e usa a forma intermediária U-code (Universal Pascal Code) [PERKINS 79]. U-code é uma versão generalizada de P-code. P-code é uma linguagem intermediária para uma máquina abstrata à pilha chamada "P-machine". U-code incorpora P-code como uma linguagem básica e inclui informações necessárias para auxiliar na produção de uma boa otimização. O redirecionamento de UGEN baseia-se na tradução de U-code para a máquina alvo usando um novo conjunto de "schematas". O processo de tradução incorpora o contexto das instruções de U-code que estão sendo traduzidas e segue o estado da máquina real usando um mapeamento entre o código U-code da máquina virtual e a máquina alvo.

A noção de "schema" ou tradução entre a forma intermediária do programa e a linguagem de máquina alvo baseado em macros tem sido muito usado em projetos de geradores de

código convencionais e redirecionáveis. Na maioria das vezes, uma simples expansão de macro baseada na forma intermediária de cada operador gera código. Entretanto, este método possui várias ineficiências, a maior delas se relaciona com a falta de contexto usado na expansão de uma instrução abstrata em particular.

A abordagem usada por UGEN difere das outras abordagens interpretativas porque nela o estado da máquina virtual grava o contexto necessário em qualquer ocasião. Uma operação em particular de U-code pode fazer com que o gerador de código altere o estado da máquina virtual e então gere instruções. Ademais, a geração de instrução também atualiza o estado da máquina virtual para corresponder ao estado da máquina real. Assim, a porção dependente de máquina do gerador de código é isolada no "schemata".

O algoritmo de geração de código funciona da seguinte maneira:

1. Usando a próxima instrução do conjunto de instruções e o estado da máquina virtual, escolhe-se um "schema" de tradução.
2. Usa-se o "schema" para emitir código, transformar o estado da máquina virtual e efetuar as mesmas transformações sobre a máquina alvo e finalmente atualizar o estado da máquina alvo, por exemplo, o conteúdo dos registradores e o código de condição.

Além disto, o algoritmo de geração de código é responsável pela alocação de registradores, etc.

A maior vantagem de UGEN está no fato de usar "schemata" para especificar a tradução e a incorporação do estado da máquina virtual no esquema de tradução. O estado da máquina virtual provê a habilidade de incorporar contexto na escolha das instruções e um método independente de máquina, para lidar com recursos da máquina real. Existem muitas desvantagens nesta técnica, uma delas é que, muito embora o "schemata" sirva, primeiramente, para isolar conceitos dependentes de máquina, alguns aspectos do algoritmo de geração de código tende a aparecer no mesmo. Este aparecimento aumenta o trabalho necessário para redirecionar o gerador de código.

3.1.2 Abordagem dirigida por reconhecimento de padrão

Por razões de portabilidade, a pesquisa em geração de código tem se concentrado na separação da descrição da máquina alvo do algoritmo de geração de código. A grande vantagem da abordagem dirigida por reconhecimento de padrão está no fato de se poder usar o mesmo algoritmo para as diferentes arquiteturas existentes.

Uma série de trabalhos são discutidos nesta seção: Weingart [WEINGART 73], Johnson, Newcomer [NEWCOMER 75], Aho e Johnson [AHO 76], Ripken [RIPKEN 77], Fraser

[FRASER 77], entre outros. Alguns dos trabalhos citados nesta relação já foram descritos em seções anteriores, como por exemplo, o de Aho e Johnson; o de Newcomer, isto porque, na verdade, eles se enquadram em ambas abordagens. Todos eles formalizam o processo de geração de código no sentido da separação entre o algoritmo de geração de código e as tabelas dependentes de máquina com as quais eles operam. Eles diferem, porém, na generalidade da representação de máquina e na assistência fornecida na construção de tabelas.

Abordagem de Weingart: Segundo Ganapathi et alii [GANAPATHI 82b], Weingart em seu trabalho, "An efficient and systematic method of compiler code generation", introduz a técnica de reconhecimento de padrão para evitar a interpretação. Em seu método, características da máquina alvo são codificadas em uma simples árvore de padrões supostamente um veículo compacto e eficiente de representação da maior parte das informações dependentes de máquina. O gerador de código é um caminhador de árvore que aceita unidades básicas (tokens) da árvore sintática da linguagem fonte e as armazena até que um casamento adequado possa ser encontrado na árvore de padrões.

Para transportar este esquema de geração de código para outra máquina, basta criar uma nova árvore de padrão para a nova máquina. Porém, na prática, as idéias de Weingart não são muito fáceis de implementar; primeiro porque a criação de uma única estrutura de árvore para codificar todos os padrões de instruções em potencial e sequências de códigos é frequentemente difícil; segundo, porque existe a possibilidade de que em algumas arquiteturas não haja instruções que casem com a árvore sintática. Falhas no reconhecimento de padrão são efetuadas por uma série de conversões de padrões. Entretanto, não há maneira de determinar se um conjunto suficiente de conversões de padrão foi fornecido. Como consequência, o gerador de código pode não conseguir produzir código para alguma subárvore válida da linguagem fonte. E por último, algumas arquiteturas provêem uma série de instruções para implementar uma única construção da linguagem fonte. A qualidade do código depende criticamente da seleção da instrução mais apropriada, por exemplo, usar uma instrução de incremento ao invés de uma instrução para somar a constante um. Isto requer um cuidado especial na escolha da melhor instrução. A técnica proposta por Weingart não pode fazer esta escolha.

Abordagem de Ripken: Ripken, segundo Ganapathi et alii [GANAPATHI 82b] usa uma versão estendida do algoritmo de Aho e Johnson apresentado na Seção 3.1 para gerar um código "ótimo" a partir de árvores de expressão. Em seu método, a representação intermediária consiste em uma árvore de expressões com atributos conectados juntos como em um grafo de acordo com o fluxo de controle do programa fonte. O conjunto de instruções da máquina é descrito como uma árvore de padrão com atributos e com um conjunto de regras de transformação de atributos. Um pré-passo para a geração de código mapeia os tipos simples da linguagem fonte para o domínio de valores característicos dos endereços da máquina.

A geração de código em si consiste na transformação da representação intermediária, o que é feito em duas fases. Na primeira fase as regras de transformação de atributos, derivadas da análise das regras de transformação de atributos da máquina são usadas para gerar código para as árvores de expressões. É assumido existir uma operação de máquina para cada operador e seus valores de atributos presentes na representação intermediária. Um esquema de caminhamento na árvore feito em três passos determina a ordem de aplicação das regras de transformações de atributos e também que regra de transformação de atributos deve ser aplicada a cada nodo.Como em Aho e Johnson, a primeira fase do esquema de Ripken enfatiza a otimização local de código. A segunda fase arranja linearmente tais blocos de códigos otimizados localmente e gera as instruções necessárias de desvios entre eles.

A diferença entre o algoritmo de Ripken e o de Aho e Johnson [AHO 76] é que no algoritmo proposto por Ripken considera-se o conjunto de instruções reais com várias classes de registradores e modos de endereçamento. As operações de transferência entre localizações de memória são consideradas juntamente com registradores, alocação e assinalamento de temporários.

No esquema proposto por Ripken, as localizações de memória são descritas como pares contendo uma classe de operando e endereço, por exemplo, (bytes, 15), (words, 16), (register, 2). Um operando é descrito pelo seu descritor de endereço e sua classe de valor. As operações são descritas por árvores de padrões, existindo pelo menos um padrão para cada representação intermediária do operador, com predicados sobre os valores dos atributos e regras de avaliação para descrever a semântica. Modos de endereçamento também são representados como árvores de gabaritos (templates). Eles são inseridos em posições apropriadas quando aplicável aos operandos na árvore da representação intermediária antes da geração de código.

Ripken não implementou sua proposta. Uma implementação direta poderia exigir um volume considerável de computação das diferentes permutações, com possibilidades de explosão combinatória. Ademais, um gerador de código baseado neste modelo poderia ser muito lento.

Abordagem de Fraser: A abordagem de Fraser [FRASER 77] é baseada no conhecimento. Em seu gerador de gerador de código, XGEN, ele usa regras "ad hoc", as quais são codificadas como subrotinas na linguagem MLISP [SMIT 70] para minimizar as dependências de máquina na geração de código. Ele usa XL, uma representação intermediária independente de máquina que pode ser adaptada para acomodar novas linguagens fontes ou máquinas alvo. As regras são usadas para efetuar alocação de memória, e neste processo, XL é re-escrita em ISP' [WICK 75], uma versão modificada de ISP [BELL 71]. A geração de código então consiste em casar esta forma em ISP' com padrões de instruções da máquina (machine instruction patterns), que também estão escritos em ISP'. A falha no casamento de padrões invoca regras, subrotinas escritas em MLISP, que tentam re-escrever a forma ISP' da representação intermediária. Exemplos de tais regras

incluem: alterar o fluxo de controle, trocar referências indiretas por indexadas, etc. As regras não garantem que uma sequência de código eventualmente seja encontrada.

Fraser efetua análise sintática de descrições ISP em tempo de geração de código para reconhecer operações a pilha, macros que atualizam códigos de condição, registradores indexadores e acumuladores. As regras especificadas como subrotinas em MLISP são usadas para alocar memória para variáveis e classificar registradores como indexadores e acumuladores. Exemplo de tais regras são:

> *"If a single instruction can add a register to some offset and use the result to index some memory then the register is an index register".*
>
> *"Store integers in the widest possible memory that can participate in an add instruction".*

Em geral, as regras de Fraser são "ad-hoc" e específicas para uma máquina. Em arquiteturas desprovidas de registradores indexadores, por exemplo, Intel 8080, as regras de registradores indexadores são inúteis. Em máquinas tais como IBM-360 ou PDP-11, onde inteiros pequenos podem ser armazenados em meia-palavra ou um "byte", a regra de alocação para inteiros é ineficiente. Descrições de máquina poderiam ser usadas em substituição a algumas destas regras porque não é tão difícil para o usuário especificar registradores indexadores e acumuladores como parte da descrição da máquina.

Na abordagem de Fraser:

1. As regras comprometem generalidade por eficiência. Elas são baseadas em observações de que muitas das arquiteturas de computadores são similares em projeto.

2. Não é possível usar as mesmas regras para arquiteturas distintas, frequentemente, é necessário regras totalmente novas. Algumas regras como:

   "Load nonaccumulators operands into accumulators"

   podem afetar potencialmente outras partes do compilador, tais como, alocação de registradores. Além disto, em seu esquema, é inevitável cargas e armazenamentos redundantes.

3. É difícil utilizar instruções especiais e modos de endereçamento da máquina alvo. Por exemplo, "a = a + 1" no PDP-11 pode casar com várias instruções como "add #1, a" e "inc a". Não é claro se seu gerador de código resolveria este casamento múltiplo escolhendo a melhor alternativa.

4. O gerador de código é muito lento. A implementação em LISP existente no PDP-10 gera uma linha de código assembler a cada segundo.

Abordagem de Jonhson: Johnson [JOHNSON 78] usou uma série de idéias de Snyder na implementação do compilador C portátil [GANAPATHI 81a]. Ele propõe um esquema de geração de código baseado em gabaritos (templates) de mapeamento de árvores de programas em instruções de máquina. Os gabaritos especificam:

1. O operador da sub-árvore.
2. A localização do resultado na máquina alvo.
3. O modo de endereçamento da máquina e o tipo de dado dos operandos das expressões da linguagem, isto é, modos de registradores e tipos de ponteiros.
4. Os recursos necessários; número de temporários e registradores necessários para implementar uma sub-árvore.
5. Uma regra de reescrita (rewriting rule) especificando como trocar uma sub-árvore por outra.
6. As instruções de máquina a serem emitidas quando ocorre um casamento.

O algoritmo "template-matching" tenta casar uma sub-árvore com o gabarito apropriado em uma tentativa de transformar a sub-árvore. Tal transformação deve considerar o local do resultado especificado no gabarito. Para uma implementação eficiente do algoritmo é essencial restringir a pesquisa por um gabarito aceitável. Um gabarito casa uma sub-árvore quando todas as especificações de gabarito do item 1 ao item 5 relacionados acima são satisfeitas. O itém 4 inclui uma chamada ao alocador de recursos; o casamento falha se ele não é capaz de alocar o recurso requerido. Numa falha, uma tentativa é feita para transformar a sub-árvore usando "default" ou regras de reescrita dependentes de máquina. Por exemplo,

a += b fica a = a + b
x ++ fica ((x += 1) -1)

As deficiências da abordagem de Johnson são:

1. Gabaritos não são os únicos lugares onde a seleção de código é especificada. Outras fases do compilador devem emitir código para alocação de registradores e prólogos de subrotinas.
2. A representação intermediária é especificamente projetada para a linguagem C. Tipos de dados dependentes da linguagem são embutidos nos gabaritos.
3. Em uma falha, regras de reescrita dependentes de máquina invocam o gerador de código para uma possível alteração na árvore. Tais regras podem gerar "loops" infinitos.

4. Não é dada a devida atenção a otimizações específicas de máquina.

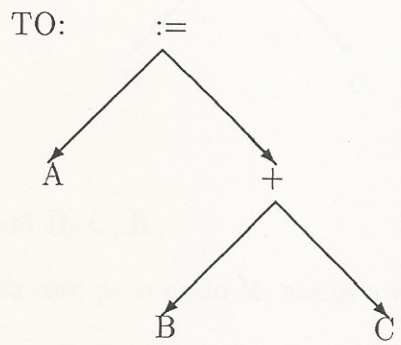
5. Mais ou menos um terço do gerador de código necessita ser re-escrito para fazer o transporte.

3.1.3 Abordagem dirigida por tabela

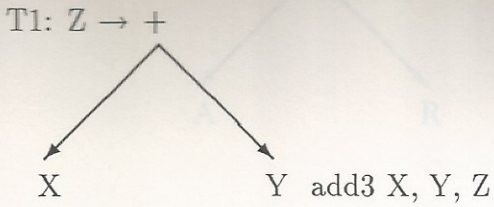
Para abranger uma variedade de máquinas-alvo é necessário haver um certo afinamento e flexibilidade no algoritmo de geração de código. Pesquisas têm se concentrado em prover esta flexibilidade através de uma análise automática da descrição formal da máquina alvo. A abordagem dirigida por tabela tenta desmembrar o problema da geração de código em três partes: alocação de registradores, alocação de memória e seleção de instruções. Entretanto, estes problemas se interagem fortemente. Por exemplo, a alocação de registradores depende da escolha das instruções e vice-versa.

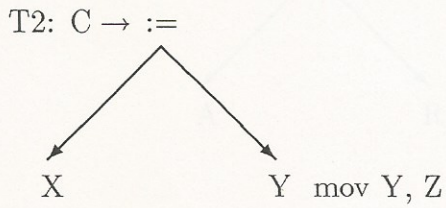
No esquema de geração de código dirigido por tabela, reconhecimento de padrão é usado para substituir a interpretação por análise de caso. Padrões de instruções podem ser representados como estrutura de árvores, [CATTELL 78], [CATTELL 80], [WULF 80], [LEVERETT 80], [KRUMME 82], [YATES 88]; ou como estrutura linear, usado por Glanville e Graham, [GRAHAM 82] e Ganapathi e Fischer, [GANAPATHI 82b]. Igualmente, reconhecimento de padrão é efetuado por análise sintática, [GANAPATHI 82b], [GANAPATHI 85], [GRAHAM 82], [CRAWFORD 82] ou por pesquisa heurística, [CATTELL 78], [CATTELL 80].

A idéia básica do método baseado em análise sintática é efetuar o reconhecimento de padrão em árvores com o auxílio de analisadores sintáticos "bottom-up". O casamento entre os gabaritos e uma árvore de expressão é muito similar ao casamento entre produções de gramática e a sequência de "tokens" durante a análise sintática. O gerador de código é organizado como um analisador sintático e a descrição da máquina faz o papel da gramática. Para ilustrar esta idéia, considere o exemplo:



padrões da máquina:

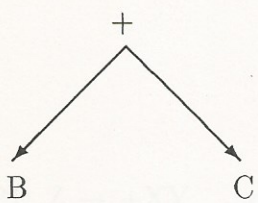




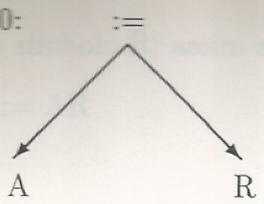
Para gerar código, sub-árvores são casadas até que o nodo nulo ($\in$) seja produzido, conforme ilustra o exemplo a seguir:

Geração de código por reconhecimento de padrão

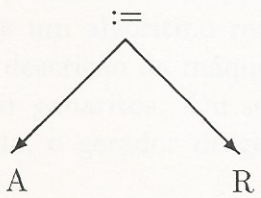
1. casa T1 com a sub-árvore



2. emite código add3 B, C, R
3. substitui a sub-árvore pelo nodo R, assim a entrada TO fica



4. casa T2 com a sub-árvore



5. emite código mov R, A

6. substitui a sub-árvore pelo nodo $\in$, assim a entrada TO fica $\in$.

Para reformular o problema de reconhecimento de padrão em termos de análise sintática, a estrutura de árvore na forma intermediária é linearizada para a forma intermediária polonesa prefixada. Padrões de árvores são então vistos como simples produções de gramática. Considere a seguinte entrada para o gerador de código

```
PO:   := A + BC
```

Produções da máquina

P1: Z → +XY add3 X, Y, Z
P2: C → :=XY mov Y, X

Para gerar código, as produções da gramática são casadas até que um não-terminal nulo ($\in$) seja produzido.

Geração de código por meio de analisador sintático

1. casa P1 com o "string" +BC

2. emite código: add3 B, C, R

3. substitui o "string" pelo símbolo R; assim a entrada PO fica :=AR

4. casa P2 com o "string" := AR

5. emite código mov R, A

6. substitui este "string" pelo símbolo $\in$; assim a entrada PO fica $\in$.

Abordagem de Cattell: Cattell [CATTELL 78], [CATTELL 79], [CATTELL 80], Wulf [WULF 75], [WULF 80] usam TCOL [BROSGOL 80] uma representação intermediária baseada em árvore e um algoritmo recursivo para percorrer a árvore e gerar código. Em seus trabalhos, a descrição da máquina é mais complexa e é necessário uma análise da mesma para derivar gabaritos. Ou seja, neste caso tem-se, além do gerador de código orientado por tabela, o gerador de gerador de código que deriva as tabelas. Gabaritos, da forma:

"tree-pattern $\Rightarrow$ result-sequence"
(LHS) (RHS)

são usados para especificar a tradução de uma árvore de programa em TCOL para código de máquina. A sequência resultante especifica o código a ser gerado, as chamadas ao alocador de registradores ou geradores de rótulos ou demais casamentos que serão recursivamente efetuados. O lado direito destes gabaritos, "result-sequence", especifica o código a ser gerado, ou futuros "submatches" a serem efetuados, quando uma árvore de padrão (tree-pattern) é encontrada na árvore de programa. Gabaritos são agrupados em "schemas". Um "schema" é um conjunto ordenado de gabaritos. Os "schemas" representam contextos diferentes nos quais código pode ser gerado, tais como:

1. Resultado do fluxo é necessário, por exemplo, um desvio condicional.

2. Um valor é requerido como resultado, por exemplo, inteiro, real, booleano, etc.

3. Nenhum resultado é desejável, por exemplo, uma árvore de comandos.

4. Um modo de endereçamento deve ser selecionado.

O lado esquerdo destes gabaritos, "pattern tree", é uma árvore cujos nodos interiores são aqueles que possuem um ou mais filhos, e nodos folhas são aqueles que não possuem filhos. Os nodos interiores representam operadores da linguagem, nos quais o casamento (match) deve ocorrer. Os nodos folhas representam um conjunto de modos de acessos, ou seja, um conjunto de acessos de uma classe de operandos.

Por exemplo, um gabarito pode ser:

$$R \leftarrow R + E \Rightarrow ADDR, E$$

onde, R e E representam a classe dos operandos e o lado direito consiste em uma instrução de adição "ADD" a ser emitida.

O gerador de código percorre a árvore de programa começando pela raiz, numa tentativa de casar cada nodo com os padrões do lado- esquerdo (LHS) dos gabaritos na tabela. Quando um padrão casa, o lado-direito (RHS) do gabarito correspondente é processado.

Os gabaritos devem ser compostos recursivamente para casar toda a árvore de programação. A falha no casamento de operandos (operand mismatches) são forçosamente resolvidos pela "subtargeting operation", a qual consiste em alocar uma posição para o tipo de dado e emitir um "move" para aquela posição. "Move" neste contexto, não se refere ao código de operação da máquina alvo e sim, a uma sequência de instruções que efetuam a operação de "move" necessária. Por exemplo, a atualização do código de condição é também considerado como uma operação "move", onde o gerador de código muda o resultado para o registrador de código de condição. Se um operador da árvore de programa não casa com nenhum operador do gabarito, é feita uma tentativa para transformar este operador usando a árvore de equivalência de axiomas (Veja Seção 4.1.3) e uma pesquisa heurística. Casamentos múltiplos são resolvidos classificando as alternativas em ordem decrescente de preferência e escolhendo a primeira. Por exemplo, $x \leftarrow x +$ constante.

Muito embora o trabalho de Cattell seja mais geral que o modelo de Newcomer, que trata somente com expressões aritméticas, ele possui as seguintes deficiências:

1. O gerador de código evita problemas sobre dependência de máquina, tais como, alocação de espaço e endereçamento de variáveis de diferentes formatos.

2. Gabaritos fazem parte do algoritmo de geração de código porque algumas sequências de resultados (result-sequences) especificam casamentos adicionais a serem efetuados. Portanto, é difícil alterar os gabaritos sem mexer no algoritmo.

3. Para "subtargeting" ter sucesso, deve haver instruções de "move" na máquina alvo entre todos os tipos de localizações possíveis. Do contrário o gerador de código poderá bloquear a geração de código para uma árvore de programa válida. As vezes para prevenir o bloqueamento, haverá "moves" desnecessários.

4. A falha no casamento entre operadores invoca uma pesquisa heurística que é recursiva e pode ocasionar uma explosão combinatória. A pesquisa deve parar em algum ponto para que o gerador de código não entre em "loop" ou não gaste muito tempo. Como consequência disto, nenhum código de máquina pode ser gerado em casos, onde a pesquisa é interrompida.

5. Sequências "ótimas" de código não são normalmente produzidas. Casos especiais de inclusão de operações tais como auto- incremento são difíceis de descrever como

gabaritos. Além disto, localizações equivalentes não são reconhecidas. Assim, se um registrador contém um operando que também está em uma posição de memória, o gerador de código falha ao identificar esta equivalência e usa o registrador.

6. Casamentos múltiplos são "estatisticamente" resolvidos pela ordenação das alternativas. Esta estratégia, muitas vezes não resulta em boas sequências de código.

Abordagem de Glanville e Graham: A abordagem do trabalho de Glanville e Graham [GANAPATHI 82b], muito embora não seja a solução ideal, é considerada como um grande avanço em relação a automatização da geração de código em compiladores. O código gerado é razoável e o gerador de código é quase completamente redirecionável. É utilizado uma representação intermediária de mais baixo nível na forma de expressões polonesa prefixada. A alocação de memória e "binding" são assumidos já terem sido feitas por outras fases do compilador. O algoritmo de geração de código é derivado da teoria de análise sintática baseada em gramáticas livres do contexto [AHO 73]. Instruções da máquina alvo são também expressas na forma prefixada. As instruções são descritas como produções de gramática com o seu lado esquerdo (LHS) especificando o resultado de uma operação e o lado direito (RHS) a operação. Uma instrução em assembler computando o lado direito é suprida em cada produção. Assim, por exemplo, r.1 $\rightarrow$ + r.1 k = 1 "INC 1" especifica que uma soma de 1 ao registrador 1 pode ser obtida por uma instrução "INC r1".

Um mapeamento um-a-um é assumido entre as instruções da máquina alvo e as produções que servem como gabaritos da máquina para o gerador de código. Como o modo de endereçamento dos operandos são explícitamente descritos como símbolos terminais da gramática, esta restrição um-a-um é essencial. Este método usa uma gramática livre do contexto para descrever a linguagem intermediária acoplada a um esquema de tradução dirigida por sintaxe. Associada a cada produção da gramática, existe uma sequência de gabaritos que especifica a tradução de código intermediário para código de máquina. O gerador de código faz uma análise sintática "bottom-up" sobre a forma intermediária e a cada redução emite as sequências de instruções de máquina apropriadas. O avaliador de padrões (pattern-matcher) é, portanto, um reconhecedor "shift-reduce" dirigido por tabela, sendo a tabela do analisador sintático construída automaticamente a partir de uma descrição de máquina por um algoritmo similar ao de um gerador LR(1), dado que gramáticas são quase sempre ambíguas. Este esquema enfatiza a exatidão do código gerado. Casamentos múltiplos produzem conflitos "shift-reduce" ou "reduce-reduce" que são resolvidos heuristicamente. Conflitos "shift-reduce" são resolvidos em favor do "shift" de tal forma que as instruções simples, porém mais poderosas são preferidas às sequências de instruções equivalentes. Da mesma forma, os conflitos "reduce-reduce" são resolvidos em favor de produções com o maior lado direito. No caso de conflitos entre produções do mesmo comprimento, um ordenamento "melhor instrução primeiro" é usado para selecionar a primeira produção.

Na descrição da máquina de Glanville, em expressão polonesa prefixada, os diferentes

tipos de dados da máquina alvo, tais como: "bytes", "words", ponto flutuante e modos de endereçamento especial, tais como, autoincremento e autodecremento não são usados. Por exemplo na produção r.1 → +r.1 r.2 "ADD r.2, r.1", os tipos de dados dos operandos e aqueles do resultado não são evidentes.

Descrições de implementações de geradores de códigos baseados no modelo de Graham-Glanville podem ser encontradas nos artigos de Graham et alii [GRAHAM 82], Bird [BIRD 82], Landwehr et alii [LANDWEHR 82] e Crawford [CRAWFORD 82]. Graham et alii discutem a implementação de um gerador de código local para o VAX-11. Bird discute a implementação de um gerador de código para um compilador Pascal em um AMDAHL-470. Landwehr et alii discutem sua implementação para MOTOROLA MC-68000 e para SIEMENS-7000. E Crawford discute a implementação de um gerador de código como uma parte do compilador Pascal-86 desenvolvido pela Intel Corporation, como um dos vários compiladores suportando a família dos microprocessadores IAPX-86 (8086). A linguagem suportada pelo compilador é uma extensão do "Draft International Standard Pascal" nível 0.

Muito embora o esquema de Glanville seja eficiente, e provavelmente correto, ele não é de todo portátil por uma série de razões:

1. A representação intermediária IR é em muito baixo nível; ela contém suposições sobre a estrutura de endereçamento da máquina alvo. Mapeamento entre operador na representação intermediária e o código de operação na máquina alvo é requerida ser um-a-um. Assim, ao transportar um compilador de uma máquina para outra, trocas devem ser feitas na representação intermediária.

2. A solução heurística para casamentos múltiplos falha em certos casos, por exemplo, na escolha de instruções de dois ou três endereços no VAX-11/780. Muito embora tais casos possam ser resolvidos usando semântica para controlar o analisador sintático.

3. O gerador de código não se preocupa com retenção de informação, ou seja, valores deixados nos registradores oriundos de computações anteriores. Assim, eliminação de armazenamentos e carregamentos redundantes, reconhecimento de posições equivalentes, inclusão de adição ou subtração via auto-incremento e decremento não são feitos.

Abordagem de Ganapathi: O método proposto por Ganapathi em sua dissertação de doutorado em 1980, cuja descrição se encontra também em [GANAPATHI 82a], é uma extensão e um sucessor natural do trabalho de Glanville. Gramáticas de atributos são usadas para especificar a tradução a partir de uma representação linear da árvore sintática, para uma representação em código objeto de programas.

Gramáticas de atributos foram propostas por Knuth em 1968 como uma maneira de

especificar formalmente semântica no escopo de gramáticas livres de contexto de uma linguagem de programação (GLC). Intuitivamente, cada símbolo da gramática em uma GLC é permitido ter um número fixo de valores associados, chamados atributos, cujo domínio pode ser finito ou infinito. À medida que um trecho da entrada é analisado, os atributos são avaliados. A árvore sintática resultante representa a semântica do trecho da entrada. Os atributos associados a um símbolo podem ser sintetizados ou herdados. Os atributos sintetizados são usados para passar informações para cima na árvore sintática. Os atributos herdados são usados para passar informações para baixo na árvore sintática.

Na abordagem de Ganapathi, o algoritmo básico de Glanville é modificado para prover um processamento de atributos formalizado e um guia semântico automático, [GANAPATHI 82b]. As principais extensões introduzidas estão relacionadas com a estrutura do gerador de código, atribuição de endereços, otimizações dependentes de máquina e redirecionamento. Atributos semânticos e atributos predicados são usados para atingir este objetivo. A máquina alvo é descrita usando gramática de atributos ao invés de gramáticas livre do contexto [AHO 73] e a geração de código é efetuada pelo analisador sintático com a avaliação dos atributos.

A representação intermediária, chamada IR [GANAPATHI 81b] é composta de uma notação linear polonesa prefixada com atributos. IR é composta de operadores, operandos e atributos. Os operandos e operadores constituem a estrutura básica da família de IR. Atributos são acrescentados nas IR com o objetivo de propagar as informações semânticas essenciais para a atribuição de endereços, isto é, tipo, tamanho, escopo de uma variável e otimização de código. Tipos de dados não são atributos dos operadores, mas sim atributos dos operandos, portanto, "+AB" onde A é um tipo inteiro e B um tipo de dado de ponto flutuante , é um "string" correto na IR. A vantagem deste esquema é que somente um operador é necessário para qualquer tipo de operação com tipos de dados diferentes.

A representação intermediária é visualizada como uma demarcação ou linha divisória entre as partes independentes e as partes dependentes de máquina, de um compilador. O "front-end" do compilador traduz uma linguagem de programação para a representação intermediária e todas as características dependentes de linguagem são processadas por ele. Por exemplo, na linguagem ADA um operador pode ter vários significados válidos dentro de um mesmo escopo. Estes significados devem ser determinados pelo "front-end" antes que ele gere a representação intermediária. O gerador de código, "back-end", traduz a representação intermediária para código da máquina alvo. Todas as características dependentes da máquina são processada por este "back-end". Como consequência deste esquema, atribuição de endereços é tratada como a um problema de geração de código que é resolvido pelo "back-end". Entretanto o "front-end" do compilador pode influenciar na decisão da atribuição de endereço via a representação intermediária. Além disto, nenhuma suposição é feita em relação ao tipo básico da arquitetura da máquina alvo, ou seja, se a máquina é a pilha ou não. Sob este modelo de compilação, um compilador pode ser redirecionado para uma nova máquina simplesmente trocando a representação intermediária para a fase de tradução do código objeto. Como esta fase pode ser automatizada,

redirecionamento é muito mais barato do que escrever um novo compilador.

Uma das diferenças entre uma representação intermediária e as linguagens para geração de código citadas na Seção 3.1.1 é que usando uma IR os "m" tradutores citados nessa Seção podem ser substituídos por "m" descrições de máquinas, juntos com um único algoritmo de geração de código [GANAPATHI 81b]. Esta separação é essencial para isolar a descrição da máquina alvo do algoritmo de geração de código. Para uma discussão mais detalhada sobre as representações intermediárias ver Ganapathi et alii, [GANAPATHI 81b], [BROSGOL 80].

Para determinação de concordância de padrão e seleção de instrução, o conjunto de instruções da máquina alvo é descrita como uma gramática de atributos formada por três classes de produções:

1. Produções para modos de endereçamento Descreve o modo de endereçamento na máquina alvo.

2. Produções para seleção de instruções Descreve os códigos de operação com restrições para operandos para cada tipo de dado da máquina. Cada produção é da forma LHS $\rightarrow$ RHS, onde LHS é um não-terminal usualmente aparecendo com um atributo sintetizado e o RHS contem:

   (a) terminais com atributos sintetizados,

   (b) não-terminais com atributos sintetizados,

   (c) predicados para resolver ambiguidades com atributos herdados, e

   (d) símbolos de ação com atributos sintetizados e herdados.

   O lado direito de cada produção especifica um padrão na representação intermediária e a correspondente sequência de código que deve ser emitida em um casamento. O lado esquerdo de cada produção especifica uma localização explícita para conter o resultado, neste caso ele contém o tipo de dado do resultado, ou a localização do código de condição.

3. Produções de Transferência Descreve conversões para os tipos de dados da máquina para todos os outros tipos de dados.

Estas produções formam a entrada para o programa que gera o gerador de código para a máquina alvo.

Cada produção livre do contexto tem a ela associada um conjunto de regras de avaliação de atributos. Todas as regras podem ser agrupadas em símbolos predicados (predicate symbols) ou símbolos de ação (action symbols), os quais computam novos valores de atributos. Para avaliar um símbolo de ação primeiramente faz com que seus atributos

herdados estejam disponíveis. Os símbolos de ação são então aplicados e seus atributos sintetizados são calculados como resultado.

O algoritmo de geração de código proposto por Ganapathi é modularmente dividido em duas fases, a fase de atribuição de endereços e a fase de seleção de instrução. Ao nível do IR, as variáveis são representadas por seus nomes, as quais são convertidas para endereços de máquina antes das instruções serem selecionadas. Após a fase de atribuição de endereços, nomes em IR são transformados em endereços de máquina com modos de endereçamento independentes de máquina que permitem níveis multiplos de indireções e indexações. Esta expansão é o ponto crucial do mapeamento entre a gramática em IR e a gramática de atributo para a arquitetura alvo. O mapeamento desses modos de endereçamento independentes de máquina para os modos de endereçamento existentes na máquina alvo são feitos durante a análise sintática das produções de modos de endereçamento da máquina alvo.

As deficiências desta técnica são as seguinte:

1. A introdução de representações intermediárias (IR) mais complicadas, pelo uso de gramáticas de atributos, introduz maior complexidade na descrição do gerador e otimizador de código.

2. O fato de gerar código em um simples passo "bottom-up" exclui otimizações iterativas dependentes de máquina [WULF 75]. A avaliação restrita de atributos e fluxos pode afetar a qualidade do código objeto.

3. O assinalamento de registradores é guiado por atributos na IR Enquanto este esquema evita a liberação de registradores, em muitos casos, ele não garante nenhuma condição sobre eles. Um passo separado de alocação de registradores é necessário para guiar o gerador de código.

4. A maior parte das otimizações não são estendidas além de blocos básicos.

Abordagem de Yates e Scwartz: Projetistas de compiladores até pouco tempo vinham evitando o uso de programação dinâmica [AHO 86] para guiar na seleção de instruções, temendo um crescimento exponencial no tamanho do espaço de pesquisa.

Recentemente, Yates e Schwartz [YATES 88] mostraram como os benefícios da programação dinâmica são obtidos com um custo relativamente baixo. Eles apresentam um linguagem e uma ferramenta de tradução (translation tool) que automatiza parcialmente a construção de "Instruction Selection Component" (ISC) de um compilador. A linguagem de especificação é não- procedural e suporta três descrições separadas: a linguagem intermediária (IL) emitida pelo "front-end" do compilador, o conjunto de instruções (ISP) da máquina alvo e um conjunto de padrões que mapeam entre a linguagem intermediária em forma de árvore e sequências de instruções. A linguagem proposta suporta a computação

dos atributos herdados e sintetizados. Os gabaritos permitem predicados semânticos e uso de custos.

O ISC gerado usa um casamento de padrão operando em uma árvore de-baixo-para-cima e programação dinâmica para gerar um bom código local. A linguagem de especificação não faz nenhuma suposição a respeito da linguagem de implementação do gerador de código. A abordagem de seu trabalho é similar em espírito ao de Haradvala et alii [HARADVALA 84], que vê a árvore de gabaritos como regras "if-then" em um sistema baseado em conhecimento.

Em seu método, a seleção de instruções é efetuada antes da alocação de registradores. O ISC assume que um número infinito de registradores "virtuais" estão disponíveis e gera código. O alocador de registradores uma vez possuindo informações exatas sobre o ciclo de vida de cada valor, assinala os registradores físicos aos registradores virtuais. O ISC caminha na árvore IL duas vezes para cada comando. No caminhamento "top-down" da primeira vez, os atributos herdados de cada filho de um nodo são computados e passados para seus respectivos filhos. No caminhamento "bottom-up" da primeira vez, o casamento da árvore de padrões e programação dinâmica são usados em cada nodo da árvore em IL para determinar todas as possibilidades de geração de código para aquele nodo. Cada vez que um padrão casa, cria-se uma instância de uma estrutura de dados chamada "solução".

Uma solução contém os valores dos atributos sintetizados computados pelo padrão. Um dado nodo IL pode ter várias soluções. Se um nodo IL tem filhos, então cada solução para aquele nodo IL aponta para uma solução para cada filho daquele nodo IL; portanto, soluções formam uma floresta de árvores de solução.

Ao retornar à raiz da árvore IL, pode haver mais de uma solução. Neste caso, escolhe-se a melhor. Na caminhada "top-down" da segunda vez, agora sobre a árvore de solução, a escolha do "melhor" pai é propagado para seus filhos. Na caminhada "bottom- up" da segunda vez, a sequência de instruções fornecida em cada solução é emitida.

Uma solução "melhor" é aquela com menos "bytes" de código gerado. Se duas ou mais soluções têm códigos do mesmo tamanho, então aquele que faz menos referências à memória é escolhido.

Abordagem de Krumme e Ackley: Krumme e Ackley, [KRUMME 82] escreveram um gerador de código para o computador DEC-10 baseado na pesquisa exaustiva e não em heurística. Seu método é dirigido por tabela, onde as informações específicas da máquina alvo são isoladas nas mesmas. O gerador de código dirigido por tabela consiste em um interpretador e um conjunto de tabelas chamadas tabelas de códigos (code tables).

Cada nodo interior de uma árvore de expressão corresponde ao código intermediário do operador. E cada operador corresponde a uma operação sobre operandos e de um tipo específico, tal como multiplicação por inteiros longos. Em muitos casos porém, um operador

pode ser usado em várias combinações de tipos de operandos, por exemplo, em adições envolvendo inteiros com sinal e sem sinal. Cada operador corresponde a uma única tabela de código, a qual dirige a tradução do operador.

Cada tabela de código expressa um conjunto de maneiras possíveis de se traduzir o operador correspondente. As possibilidades incluem instruções de máquina diferentes, ordem de avaliação e alocação de registradores.

O método proposto por eles traduz cada nodo interior da árvore de expressões em várias sequências de instruções viáveis. Estas sequências são então colocadas juntas, formando um conjunto de tradução para uma expressão inteira. Este conjunto é então pesquisado com o objetivo de encontrar uma alternativa mais barata. Em seu método, existe a correspondência um-a-um entre as instruções na tradução e a expressão original, por exemplo, se há uma adição em uma expressão, então deve haver uma adição em algum lugar no código gerado.

Abordagem de Crawford: Crawford [CRAWFORD 82] propôs um gerador de código formado pela união dos melhores aspectos das três técnicas de geração de código: Graham-Glanville, geração de código acoplada a um esquema de tradução dirigida por sintaxe; Johnson, o mecanismo de alocação e liberação de registradores do compilador C portáil; e Wilcox, o expansor de gabaritos.

Em sua abordagem, o método de Graham-Glanville foi modificado para usar um analisador sintático LALR padrão e um construtor de tabelas, o método de alocação de registradores foi estendido sob vários aspectos com o objetivo de fazer melhor uso de uma máquina com registradores caracterizados.

O gerador de código de Crawford é dividido em três partes principais, um analisador sintático, um alocador de registradores e um expansor de gabaritos, conforme ilustrado na Figura 1.

O analisador sintático dirige o processo de geração de código emitindo referências de gabaritos (templates references), ou seja, ele lê um arquivo intermediário contendo a árvore de programa e emite uma sequência de referências de gabaritos que são colocadas juntas formando a linguagem de máquina correspondente a linguagem fonte. Sua tarefa mais importante, diz respeito a seleção da forma de instrução, isto é, o processo de casar as formas disponíveis das instruções de máquina, como por exemplo, registrador-para-registrador, registrador-para-memória, registrador-constante, com sub-expressões na árvore de programa da entrada.

O alocador de registradores armazena estas referências de gabaritos em uma árvore de expressão. Percorre esta árvore para executar as várias alocações de registradores e sequências de expansão de gabaritos através das referências de gabaritos.

A fase de alocação de registradores interpõe-se entre o analisador sintático e o expansor de gabaritos e mantém um registro (buffer) com estrutura de árvore de referências de gabaritos. A alocação de registradores é feita em dois passos sobre a árvore de expressão. O primeiro passo é efetuado de baixo-para-cima efetuado enquanto a árvore de expressão é construída. O segundo passo é uma travessia da esquerda para a direita através da árvore de expressão. Ações semânticas ocorrem em ambas visitas prefixada (de-cima-para-baixo) e posfixada (de-baixo-para-cima) do segundo passo. A maior parte da alocação de registradores é feita durante a construção da árvore de expressão no primeiro passo. Este passo calcula o conjunto de registradores necessários para computar cada sub-expressão, sub-árvore, juntamente com o conjunto de localizações onde o resultado da sub-árvore pode ser colocado.

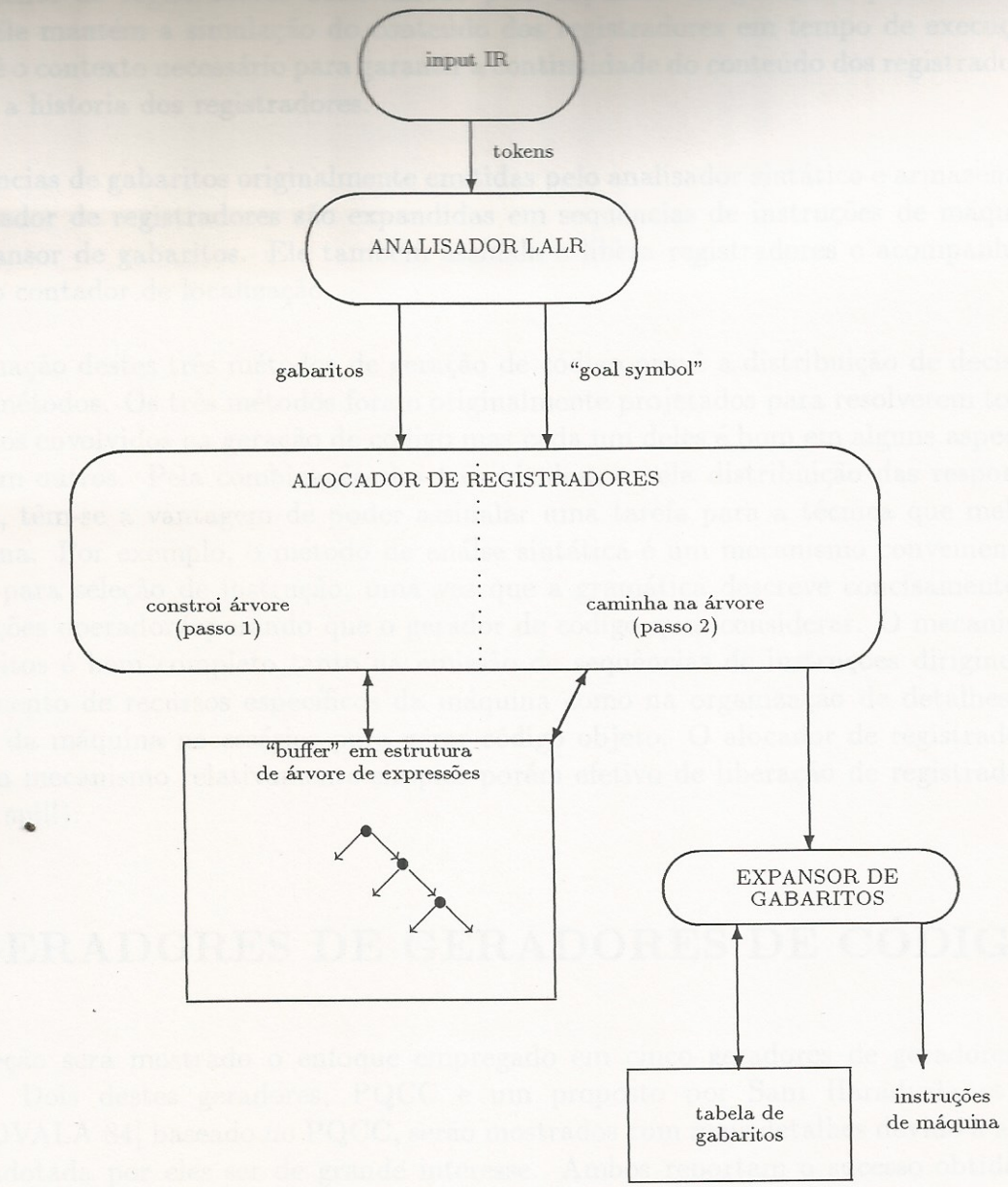


Figura 1: Estrutura do gerador de código.

O caminhamento na árvore no segundo passo em ordem prefixada serve para propagar as informações com respeito a preferência de registradores determinada no primeiro passo, para baixo na árvore. Na visita posfixada do segundo passo, o expansor de gabaritos é invocado para processar os gabaritos para cada nodo. Esta visita também propaga

assinalamento de registradores determinado pelo expansor de gabaritos para cima na árvore. Ele mantém a simulação do conteúdo dos registradores em tempo de execução. Isto provê o contexto necessário para garantir a continuidade do conteúdo dos registradores e manter a historia dos registradores.

As referências de gabaritos originalmente emitidas pelo analisador sintático e armazenado pelo alocador de registradores são expandidas em sequências de instruções de máquina pelo expansor de gabaritos. Ele também assinala e libera registradores e acompanha o código do contador de localização.

A combinação destes três métodos de geração de código provê a distribuição de decisões entre os métodos. Os três métodos foram originalmente projetados para resolverem todos os aspectos envolvidos na geração de código mas cada um deles é bom em alguns aspectos e ruins em outros. Pela combinação das três técnicas e pela distribuição das responsabilidades, têm-se a vantagem de poder assinalar uma tarefa para a técnica que melhor a soluciona. Por exemplo, o método de análse sintática é um mecanismo conveniente e eficiente para seleção de instrução, uma vez que a gramática descreve concisamente as combinações operador/operando que o gerador de código quer considerar. O mecanismo de gabaritos é bem completo tanto na emissão de sequências de instruções dirigindo o assinalamento de recursos específicos da máquina como na organização de detalhes específicos da máquina necessários para gerar código objeto. O alocador de registradores provê um mecanismo relativamente simples porém efetivo de liberação de registradores (register spill).

4 GERADORES DE GERADORES DE CÓDIGO

Nesta seção será mostrado o enfoque empregado em cinco geradores de geradores de código. Dois destes geradores, PQCC e um proposto por Sam Haradvala et alii [HARADVALA 84] baseado no PQCC, serão mostrados com mais detalhes devido a abordagem adotada por eles ser de grande interesse. Ambos reportam o sucesso obtido na aplicação de técnicas de Inteligência Artificial no processo de geração de código de um compilador.

4.1 PQCC - Production-Quality Compiler Compiler

Em linhas gerais, PQCC [WULF 80], [LEVERETT 79] e [LEVERETT 80] é um sistema de geração de compiladores que produz compiladores capazes de gerar um bom código. A entrada para PQCC consiste na descrição da linguagem a ser compilada e na máquina alvo para a qual se deseja produzir código. E a saída de PQCC é um compilador de produção de boa qualidade, PQC. A Figura 2 mostra o diagrama de bloco de PQCC e PQC.

A estrutura de um compilador que poderia ser usado com o gerador de código descrito nesta seção é o mesmo de Bliss-11 [WULF 75].

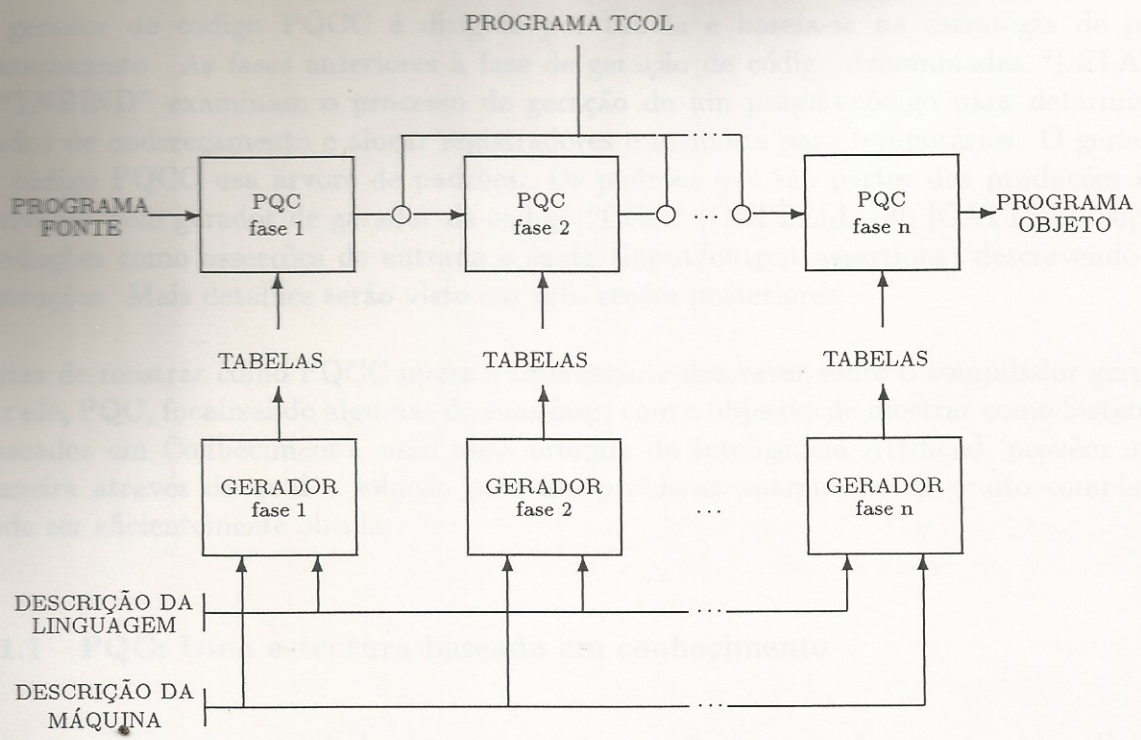


Figura 2: Diagrama de bloco detalhado do sistema PQCC.

O compilador BLISS-11 é composto das seguintes fases: LEXSYN, FLOW, DELAY, TNBIND, CODE e FINAL.

LEXSYN: Análise léxica e sintática, a entrada para esta fase é o programa fonte e a saída é uma árvore abstrata do programa em TCOL.

FLOW: Análise de fluxo. Sub-expressões são analisadas e sequências de operações são rearranjadas.

DELAY: Executa otimizações especializadas na árvore, além de determinar o "formato" do código final, ou seja, que tipo de registradores serão necessários.

TNBIND: Aloca temporários requisitados pelo programa para os tipos disponíveis de registradores.

CODE: É o gerador de código propriamente dito. A entrada é rearranjada, a árvore sintática é decorada e a saída é uma sequência de código simbólico.

FINAL: Executa otimizações locais e de desvios que não são reconhecidos até que o código adjacente seja conhecido.

O gerador de código PQCC é dirigido por tabela e baseia-se na estratégia de pré-planejamento. As fases anteriores à fase de geração de código denominadas "DELAY" e "TNBIND" examinam o processo de geração de um pseudo-código para determinar modos de endereçamento e alocar registradores e memória para temporários. O gerador de código PQCC usa árvore de padrões. Os padrões que são partes das produções são derivados pelo gerador de gerador de código "CGG" [CATTELL 78], [CATTELL 80] de produções como asserções de entrada e saída (input/output assertions) descrevendo as instruções. Mais detalhes serão visto em sub- seções posteriores.

Antes de mostrar como PQCC opera é interessante descrever sobre o compilador gerado por ele, PQC, focalizando algumas de suas fases com o objetivo de mostrar como Sistemas Baseados em Conhecimento, uma idéia oriunda de Inteligência Artificial, provêm uma maneira através da qual a solução para um problema aparentemente muito complexo, pode ser eficientemente obtida.

4.1.1 PQC: Uma estrutura baseada em conhecimento

Para o usuário, um compilador é normalmente uma "caixa preta" como mostra a Figura 3, a entrada consiste em um texto fonte e a saída é um código objeto relocável. Os livros-textos sobre compilação revelam uma visão mais detalhada, como ilustra a Figura 4. Normalmente é dito que um compilador é composto de três fases, as quais, sucessivamente transformam o programa fonte por meio de várias formas intermediárias em um programa objeto. Muitas vezes, uma quarta fase de "otimização" é também descrita, como mostrada na Figura 4 em letras minúsculas. A estrutura de um PQC entretanto consiste em um número muito maior de fases.

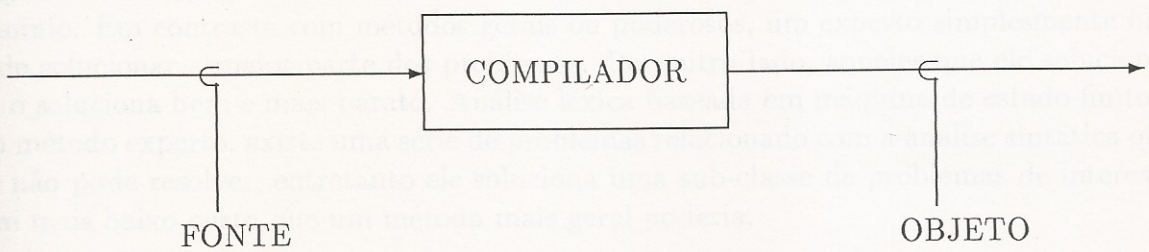


FIGURA 3: Compilador do ponto de vista do usuário.

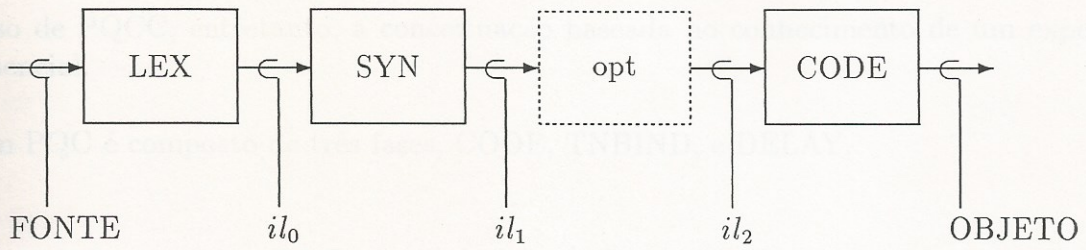


FIGURA 4: Compilador do ponto de vista de livros textos.

Em princípio, é possível usar uma técnica geral, porém, poderosa para efetuar a tradução de um compilador. É sabido que não há necessidade real para a separação da análise léxica da análise sintática e de fato a divisão entre elas em um compilador em particular é feito arbitariamente. Os métodos mais gerais usados para a análise sintática de gramáticas livres do contexto poderia ser usado para efetuar todo o trabalho referente a fase análise léxica, entretanto, a rapidez do sistema como um todo é melhorado com sua separação.

Este é um exemplo simples de um fenômeno mais geral que tem sido especialmente perceptível em pesquisas na área de Inteligência Artificial. Muitas das primeiras pesquisas em Inteligência Artificial focalizavam em métodos gerais "poderosos" para solucionar os problemas. Na realidade porém, estes métodos são muito lentos para serem práticos no domínio de tarefas complexas. Uma característica comum dessas tarefas é que elas requerem muitas espécies de conhecimento e um conhecimento profundo do domínio a ser aplicado. Métodos gerais não possuem este conhecimento de uma forma direta. Em geral os métodos gerais nunca sabem nada em particular e sabem tudo em geral. Assim, quando deparados com uma tarefa específica, métodos gerais devem "redescobrir" situações que poderiam ser óbvias a um especialista. Nestas tarefas complexas, sabe-se que um conjunto de métodos complementares ou experto é muito melhor do que os métodos mais gerais. Sistemas organizados ao redor deste esquema são usualmente referidos como sistemas especialistas. Um experto é simplesmente um algoritmo ou uma heurística com um domínio restrito de aplicação, mas o qual encorpora uma grande porção de conhecimento específico naquele domínio, e portanto é muito mais eficaz na resolução de problemas dentro de seu domínio. Em contraste com métodos gerais ou poderosos, um experto simplesmente não pode solucionar a maior parte dos problemas. Por outro lado, aqueles que ele soluciona, ele o soluciona bem e mais barato. Análise léxica baseada em máquina de estado-finito é um método experto, existe uma série de problemas relacionado com a análise sintática que ele não pode resolver, entretanto ele soluciona uma sub-classe de problemas de interesse com mais baixo custo que um método mais geral poderia.

Não é comum um projetista de compilador pensar em três ou quatro fases de um compilador como uma coleção de médodos expertos quando se está escrevendo um compilador para uma linguagem e máquina alvo específicas. Tal conceituação é desnecessária. No

caso de PQCC, entretanto, a conceituação baseada no conhecimento de um experto foi essencial.

Um PQC é composto de três fases, CODE, TNBIND, e DELAY.

4.1.2 CODE: O gerador de código experto

A estratégia básica usada no gerador de código de PQC é baseada em uma biblioteca, consistindo em pares "pattern-action". O programa que gera esta biblioteca para CODE é chamado GEN e será descrito na próxima sub-seção. Os "patterns", como a representação intermediária do programa, são árvores; as folhas dessas "pattern-trees" especificam propriedades da árvore de programa que eles casam. As "actions" são simplesmente código para serem emitidos. Por exemplo, escrevendo "pattern-trees" em uma forma prefixada na linguagem LISP, pares de "pattern-action" para o PDP-10 seria:

```
pattern:(:= $1:memory $2:register)
action: MOVEM $2,$1.
```

Cada padrão descreve uma árvore, o primeiro item é um operador no nodo raiz da árvore. Os itens restantes descrevem os descendentes daquela raiz; o símbolo dolar seguido por um dígito indica nomes para os descendentes e palavras como "register" descrevem as suas propriedades.

CODE percorre a árvore de programa e para cada nodo da árvore, ele encontra todos os "pattern-trees" da biblioteca que o casam; e então seleciona o de menor custo. Em geral, os "pattern trees" conterão mais de um nodo; isto acarreta duas implicações para CODE. Primeiro, ele deve marcar todos os nodos casados pelo padrão selecionado e não deve subsequentemente gerar código para nodos assim marcados. Segundo, e mais importante, CODE gera código "backwards". O gerador de código PQC gera a última instrução do programa primeiro. Para ver como isto é feito, note primeiramente que prosseguir "forward" através da representação do programa é equivalente a uma travessia na árvore "bottom-up" e da esquerda-para-direita; prosseguir "backwards" é equivalente a uma travessia "top-down" da direita-para-esquerda.

Por exemplo, considere a árvore para o comando de assinalamento: "i := i-j". Se a travessia "forward" fosse usada, o primeiro nodo interessante que encontraríamos seria a subtração, e código seria gerado para aquele nodo de um padrão como o mostrado acima. Como consequência não seria visto o contexto maior, no qual i aparece como o lado-esquerdo do assinalamento até muito tarde. Assim, ao invés de gerar código para computar o resultado diretamente na variável, algum tipo de temporário seria necessário e uma instrução adicional seria gerada para armazenar o temporário em i. Por outro

lado, atravessando a árvore "backwards", ou seja, "top-down", este contexto maior seria visto primeiro possibilitando a aplicação da estratégia "maximal pattern". Cattell [CATTELL 78] denominou esta estratégia de método de geração de código "maximal munch".

4.1.3 GEN: O gerador de gerador de código

A seção anterior envolveu o uso do conhecimento codificado em pares de código "pattern-action" em uma forma direta. Esta seção considera como os pares "pattern-action" são automaticamente derivados da descrição formal da máquina. Isto por vez, exporá um outro conjunto de idéias oriundas da pesquisa em Inteligência Artificial. Como mencionado anteriormente, o programa que gera os pares "pattern-action" para CODE é denominado GEN. O objetivo de GEN é converter a descrição formal da máquina nestes pares. O selecionador de código descende diretamente de CGG, o gerador de código descrito por Cattell [CATTELL 78]. Uma das diferenças entre GEN e CGG está no fato de que a alocação de registradores e determinação dos modos de acesso são feitos em fases separadas no sistema PQCC.

A entrada para GEN consiste em duas partes: a descrição da máquina e um conjunto de axiomas. Os axiomas usados são basicamente aqueles oriundos da aritmética e lógica. Por exemplo, eles incluem

$$E \equiv -E$$
$$E + 0 \equiv E$$
$$B1 \wedge B2 \equiv \neg(\neg B1 \vee \neg B2)$$

onde B denota expressões booleanas e E expressões aritmética. Além disto, pode ter para máquinas específicas, axiomas que por exemplo referem-se a valores lógicos e aritméticos. Por exemplo, em máquina com complemento de dois tem-se:

$$-E \equiv (\neg E) + 1$$

Outros axiomas capturam noções essenciais como as sequências e desvios.

A descrição formal da máquina usada por GEN contém detalhes sobre classe de armazenamentos, isto é, tipos de registradores e memória, cálculo de endereço efetivo, informações sobre custo, e assim por diante. Porém o mais importante, é que ele contém uma descrição detalhada do conjunto de instruções de máquina. Esta descrição é na forma de asserções de entrada e saída associada com cada instrução. Asserção de entrada e saída é uma idéia emprestada da pesquisa em semântica formal de linguagem de programação, e é frequentemente escrita da forma

P1 { C } P2

onde P1 e P2 são fórmulas lógicas (logical formulae), e C é algum comando em linguagem de programação. Tal fórmula é lida da seguinte maneira: se o estado do programa (variável) é de tal forma que o predicado P1 é verdade antes da execução de C, então o estado do programa (variáveis) após a execução de C será tal que P2 é também verdade. Assim, por exemplo, a formula

x= 1 { x := x + 1 } x = 2

afirma o fato de que se x tem o valor 1 antes que o assinalamento seja executado, ele terá o valor 2 logo após. Por exemplo, instruções SUB e MOV no PDP-11 podem ser caracterizadas como:

$$\{SUB\text{src dst}\}\text{dst} = \text{dst}' - \text{src} \wedge N = (\text{dst} < 0) \wedge Z = (\text{dst} = 0)$$
$$\{MOV\text{src dst}\}\text{dst} = \text{src} \wedge N = (\text{src} < 0) \wedge Z = (\text{src} = 0)$$

onde N e Z são dois dos onze bits do código de condição e o primeiro dst(dst') na definição de SUB denota o valor de dst antes da instrução ser executada.

As asserções são representadas como árvores de padrão do programa (program tree patterns) as quais correspondem a ação que a instrução executa. Neste sentido, a definição anterior de asserções de entrada e saída é equivalente a um comando condicional

$$\text{if } B_i \text{ then } L_i \leftarrow E_i \text{ para asserção i}$$

onde B_i é uma expressão booleana, L_i é a localização da expressão e E_i é um valor da expressão B_i.

A implementação de GEN consiste em duas partes. A primeira chamada SELECT, propõe um conjunto interessante de árvores de programas para as quais seria conveniente ter padrões de geração de código (code-generation patterns). Este conjunto inclui pelo menos uma árvore para cada construção da linguagem fonte, além de outras árvores que podem ter implementações interessantes. SELECT sabe por exemplo que literais e principalmente pequenos literais como, 0, 1, -1 frequentemente dão margem para casos especiais que podem ser manuseados mais frequentemente. A segunda parte, chamada SEARCH, encontra alternativas de implementações para cada uma das árvores propostas por SELECT e emite pares de padrões de ação (pattern-action pairs) para as mais eficientes.

A análise "means-ends" supre muito do conhecimento necessário pelo SEARCH, mas isto não é suficiente. Quando isto ocorre aplica-se os axiomas de lógica, para mais detalhes

ver Wulf [WULF 80]. GEN não para ao encontrar uma sequência de instrução para um dos objetivos propostos por SELECT. Ao invés disto, ele gera todas as sequências que ele pode, dentro de um limite de tempo especificado na sua pesquisa. SEARCH usa a informação de custo da instrução para diferenciar as sequências de baixo custo e estas são as emitidas para uso de CODE, TNBIND, etc. Do ponto de vista teórico, não é garantido que GEN encontre uma sequência de código ótima para as árvores propostas por SELECT. Mesmo que lhe fosse permitido executar por um período indefinidamente longo, isto seria verdade. O fato de que um limite é posto na sua pesquisa torna isto óbvio. Como em geral, conjunto de instruções são simples GEN encontra sempre sequências ótimas.

O PQCC demanda o mínimo do implementador. A linguagem de entrada TCOL foi especificada para ser independente da linguagem fonte. O GEN tem inteira responsabilidade para guiar o gerador de código a partir de uma descrição da máquina gerada automaticamente. Todos os aspectos da interface de TCOL e o gerador de código são manuseados pelo GEN, em particular, o implementador não provê informações sobre como devem ser as instruções que melhor correspondam aquelas da linguagem intermediária.

Wulf, [WULF 80] propôs construir um gerador de código especialista, mas tal não aconteceu. Ao invés disto, todo expertise sobre a máquina alvo reduziu-se na biblioteca contendo os pares "pattern-action".

4.2 Sistema Especialista para geração de código de boa qualidade

Haradvala, Knobe e Rubin, relatam em seu trabalho [HARADVALA 84], suas experiências em relação a aplicaç ão de um Sistema Especialista na produção de código objeto de boa qualidade em três projetos diferentes, envolvendo arquiteturas totalmente diferentes.

O primeiro projeto se refere aos compiladores HAL/S. HAL/S é usado para programação em computadores de bordo do ônibus espacial, é uma aplicação de tempo real na qual a eficiência do código objeto é fundamental. Muito embora os compiladores HAL/S empreguem atualmente técnicas de otimização independente de máquina, no projeto original, a qualidade do código gerado foi obtido com a dependência de máquina. Dependência esta proveniente do alto grau de conhecimento do programador de linguagem assembler. Esta façanha foi experimentalmente verificada codificando programas em HAL/S e em linguagem assembler. O gerador de código resultou em um processo extremamente caro e lento. O código objeto de HAL/S foi examinado a mão e verificou- se o que o experto havia feito em linguagem assembler. O gerador de código foi então modificado. O resultado deste projeto foi muito bom, porém o gerador de código ficou extremamente complexo.

O segundo projeto foi um sistema de tempo real baseado na linguagem Pascal para o Intel 8086. Mais uma vez, o gerador de código teve que obter todo o conhecimento que um especialista possuia sobre a matéria em si. Desta vez, entretanto, Haradhvala et alii

estavam determinados a evitar o custo e a complexidade de encaixar todos "expertise" na lógica do compilador.

Nesta mesma época, Wulf publicava com sucesso o seu trabalho PQCC. Haradhvala et alii se sentiram atraídos pelo PQCC, especialmente pelo uso de um Sistema Especialista para efetuar a geração de código e resolveram adaptá-lo para seu problema. Isto os conduziu a um gerador de código, o qual caminha em uma linguagem intermediária em forma de árvore, e usa reconhecimento de padrão para substituir sub-árvores por código objeto. Os padrões da linguagem intermediária e código objetos associados são escritos como regras de produção por um especialista. O gerador de código usa o conhecimento adquirido desta forma para pesquisar localmente por sequências ótimas de código. Ao ser identificado casos especiais, mais produções são escritas. Nenhuma troca é necessária no programa para melhorar o gerador de código. O resultado é um gerador de código de qualidade similar ao de HAL/S, porém, fácil de entender e possível de se manter.

No terceiro compilador escrito para a linguagem CHILL, o uso do sistema baseado em conhecimento foi extendido para incluir um otimizador local. Otimizadores locais tradicionais raramente são bem sucedidos simplesmente devido a complexidade de manejar tantas pesquisas de casos especiais. Haradhvala et alii moveram, em seu projeto, toda a complexidade para as produções, de tal forma que o experto tem somente que escrever produções especificando as transformações do código de máquina. Uma vez que as produções contêm todas as informações específicas da máquina o próprio otimizador é independente de máquina.

O projeto de Haradhvala et alii baseou-se no PQCC. Entretanto quando eles tentaram aplicar as mesmas idéias a máquinas mais complexas como 8086, e em linguagens mais complicadas como o Pascal estendindo e Chill, eles descobriram que não poderiam gerar produções automaticamente tão bem como um especialista. Haradhvala et alii decidiram modificar o sistema PQCC para produzir as regras de produção diretamente e incorporar conhecimento adicional nelas.

A memória de trabalho do sistema é uma estrutura de árvore, na qual os nodos são rotulados com um tipo, por exemplo, expressões, constantes. Uma regra é um par consistindo de condições e ações. Uma das condições é uma estrutura de árvore de gabaritos (tree structure templates) a qual deve ser casada. A cada regra foi associada o rótulo da raíz do gabarito.

O procedimento de geração de código possui três aspectos importantes em relação à eficiência. Um aspecto é uma ordenação de regras manualmente impostas, o segundo aspecto é o fatoramento automático para eliminar computações redundantes e o terceiro aspecto é a inferência automatica da verdade sobre condições fracas quando condições mais fortes são estabelecidas.

Para cada rótulo possível, tem-se uma ou mais produções, as quais são dadas uma ordem fixa. O experto usa esta ordem para avaliar a qualidade do código produzido para cada

produção. O experto deve garantir que a última produção na lista sempre casará ou o gerador de código será capaz de bloquear em alguns tipos de entrada. As regras são aplicadas em uma pesquisa em profundidade com cada condição avaliada logo que possível. Se um casamento falha, o sistema deve voltar atrás e tentar uma nova regra.

O sistema proposto por Haradavala et alli foi escrito em Pascal e tendo o computador IBM/370 como hospedeiro. Os dois compiladores gerados por ele estão sendo usados hoje e suas manutenções são feitas por pessoas que não foram envolvidas em seus desenvolvimentos. O mesmo não aconteceu com os compiladores HAL/S, pois para mantê-los, descobriu-se que é necessário meses antes que alguem possa se inteirar de seu funcionamento. E mesmo para indivíduos experientes, leva-se dias para poder fazer pequenas modificações.

4.3 ECS - " Experimental Compiling Systems"

O projeto ECS [LEVERETT 79], [LEVERETT 80] é endereçado aos problemas de máquina alvo e a independência de linguagens em compiladores otimizadores. Eles estão primeiramente interessados em modularização e padronização de projeto de compiladores otimizadores.

O processo de compilação em seu modelo envolve a tradução para uma linguagem intermediária (IL), quase uniforme, seguida de um "loop" no qual otimizações independentes de máquina são executadas por tanto tempo quanto for necessário, seguida por uma fase de adaptação para a máquina (machine tailoring) na qual a representação IL se aproxima mais do nível de instrução da máquina alvo.

A abordagem central de ECS evita a análise de casos especiais, usando em lugar dela poucas técnicas, porém gerais e poderosas, tais como análise de fluxo global, propagação de constante, eliminação de código morto e inclui otimizações que normalmente são verificadas somente por métodos mais especializadas, ou mesmo métodos ad hoc.

4.4 ACK - "Amsterdam Compiler Kit"

O sistema de construção de compiladores proposto por Tanenbaum et alii [TANENBAUM 83] o qual é denominado "Amsterdam Compiler Kit - ACK" pode ser visto como um "tool kit" para a construção de compiladores e interpretadores.

As principais partes do "kit" são os "front-ends", que convertem programas fontes para código EM "Encoding Machine"; otimizadores, que melhoram o código EM; e "back-ends", os quais convertem o código EM para código em linguagem assembler. O "kit" é altamente modular, assim projetando-se um "front-end" e as associadas rotinas é suficiente

para implementar uma nova linguagem em muitas máquinas. E projetando-se uma tabela "back-end" e uma tabela universal para um "assembler/linker" é tudo que é necessário para transportar as linguagens previamente implementadas para uma máquina nova.

O "back-end" lê uma sequência de instruções EM e gera código assembler para a máquina alvo. Muito embora o algoritmo seja independente de máquina, uma tabela dependente de máquina deve ser suprida para cada máquina alvo. A "driving table" define efetivamente o mapeamento do código EM para o código objeto.

Segundo Tanenbaum et alii, a principal lição aprendida com este trabalho é que a idéia de UNCOL é basicamente um bom caminho para se produzir compiladores, providenciando convenientes restrições impostas na linguagem fonte e máquina alvo. Além disto, eles acreditam que muito embora compiladores produzidos por esta tecnologia não possam ser igual aos melhores compiladores escritos a mão em termos de qualidde de código, eles são certamente competitivos com muitos compiladores existentes.

4.5 SIG

SIS, [AVISSAR 85] é o projeto de um sistema de suporte à implementação automática de geradores de código que permite sistematizar o projeto de geradores de código independentes de máquina, reduzindo assim o custo de desenvolvimento de um novo compilador para uma máquina. Este sistema se caracteriza pela automatização de várias partes da geração de código e pela facilidade de uso.

SIS envolve a definição de uma forma intermediária apropriada para a entrada de um programa fonte no gerador de código; o projeto e implementação de um algoritmo de geração de código independente de máquina e o projeto de uma linguagem de especificação de gerador de código contendo as informações dependentes de máquina.

O sistema SIG recebe como entrada uma especificação de gerador de código escrita na linguagem de especificação de Geradores de código e produz como saída um gerador de código correspondente escrito em PASCAL.

O gerador de código produzido é um programa que recebe como entrada um programa na linguagem intermediária em forma de árvore e produz como saída um programa correspondente ao da entrada no código objeto de uma dada máquina alvo. O gerador de código é baseado em gramáticas de atributos.

Uma caracteristica de SIG é a portabilidade de seu produto assim como a sua própria, o sistema é escrito em Pascal padrão e produz geradores de código também em Pascal.

O SIG é essencialmente composto de:

1. gerador ou construtor de tabela,
2. de um núcleo geral de geradores de código,
3. de um gerador de definições e declarações complementares de constantes, tipos, variáveis e procedimentos para o gerador de código e,
4. de um montador de geradores de código que concatena as várias partes constituintes de um gerador de código.

O núcleo de geradores de código consiste em um conjunto de rotinas básicas constituintes do gerador de código e de definições e declarações de constantes, tipos, variáveis e procedimentos correspondentes a estas rotinas.

SIG, portanto possui duas funções básicas:

1. geração de informações complementares ao núcleo de geradores de código já existente e,
2. montagem do gerador de código através da concatenação adequada destas informações ao núcleo.

5 OTIMIZADORES

5.1 Introdução

A otimização de código tem sido uma área de pesquisa por quase três décadas. Na década de 1960 havia alguns tradutores que efetuavam a transformação de uma linguagem fonte para outra, tais como, o tradutor da IBM de Fortran para PL/1 provendo algumas otimizações [GANAPATHI 89]. Entretanto, devido às diferenças semânticas entre as linguagens, fonte e objeto, e a falta de disponibilidade de fundamentos teóricos naquela época, estes otimizadores eram grotescos. Na década de 1970 começaram a ser estabelecidos os princípios teóricos da geração de código, mas, as aplicações práticas destas idéias apareceram quase em 1980 [CATTELL 78]. Um número de algoritmos e técnicas de otimizações foram propostos e estudados, entretanto estes algoritmos ainda não cobrem os sistemas integralmente , existindo ainda, lacunas no princípio formal da teoria de otimização de código. Como consequência, a construção de um compilador otimizante continua sendo uma tarefa difícil. Aho et alii [AHO 73], [AHO 86] e Gries [GRIES 71] entre outros, apresentam vários algoritmos para coletar informações usando análise de fluxo de dados, e para, efetivamente, usar esta informação durante a otimização. Contudo, estes desenvolvimentos só começaram a ser, efetivamente, incorporados à compiladores em meados de

1970 [WULF 75], e à geradores de compiladores no início da década de 1980 [WULF 80], [LEVERETT 79], [LEVERETT 80].

O compilador BLISS/11 responsável pela implementação da linguagem de programação BLISS no computador PDP-11 [WULF 75] é considerado, ainda hoje, um projeto importante, pois ele foi o primeiro a permitir que a análise de fluxo de dados fosse efetuada diretamente na árvore sintática, ao invés de ser no grafo de fluxo. Muito embora a maior preocupação, naquela época, fosse espaço em memória, o fator tempo também foi considerado, encontrando-se nele a descrição de várias estratégias de otimização.

As técnicas de otimização mais comuns incorporam transformações em programas nos níveis local e global. Uma transformação é local se ela pode ser efetuada tendo em vista apenas os comandos em um bloco básico [AHO 86], senão ela é denominada global. Normalmente as transformações a nível local são executadas primeiro. Um bloco básico é uma sequência de comandos consecutivos no qual o fluxo de controle entra no seu início e o deixa no final sem interrupções ou possibilidades de desvios, exceto em seu final.

As transformações globais não são, portanto, substitutas para as transformações locais, ambas devem ser efetuadas sobre o programa. Por exemplo, ao se efetuar globalmente a eliminação de uma sub-expressão comum, a atenção deve se ater ao local onde a expressão é gerada no bloco, e não onde ela é recomputada dentro do mesmo. Consequentemente, para produzir um bom código, um compilador necessita coletar informações sobre o programa como um todo e distribuí-las entre cada bloco no grafo de fluxo. O processo que efetua esta tarefa é denominado análise de fluxo de dados. A informação coletada nos vários pontos do programa podem ser expressas usando um conjunto simples de equações [AHO 86].

Uma outra técnica eficaz, que melhora o código objeto, denomina- se otimização "peephole". As otimizações "peephole" são muitas vezes reconhecedores de padrões dirigidos por tabela que operam sobre o código em linguagem de montagem produzido pelo compilador. Cada vez que uma sequência de instruções é casada com uma instrução da tabela, uma troca é efetuada entre esta e a outra sequência menor e mais rápida. Normalmente este método é dependente da arquitetura da máquina alvo. O termo "peephole" foi derivado do fato de que o otimizador, na tentativa de melhorar o desempenho do programa objeto, somente vê uma pequena janela (peephole) de instruções de máquina adjacentes sem usar nenhum conhecimento global do programa.

Muito embora tenhamos definido otimização "peephole" como uma técnica para melhorar a qualidade do código objeto, ela também pode ser aplicada diretamente após a geração de código intermediário com o objetivo de melhorar a representação intermediária.

O código no "peephole" não necessita ser contíguo, muito embora algumas implementações o exijam. É uma característica deste método que cada melhoria obtida seja uma oportunidade para outros melhoramentos. Em geral, é necessário várias repassadas no código para obter o máximo de benefícios.

As transformações características desta técnica incluem a eliminação de instruções redundantes, otimizações no fluxo de controle, simplificações algébricas e uso de instruções implementadas pelo "hardware" de uma determinada arquitetura com o objetivo de efetuar eficientemente certas operações específicas.

Um ponto indispensável na geração de otimizadores é a análise formal da arquitetura da máquina alvo. Até 1960, a fase de otimização no processo de compilação era a menos entendida devido a sua total dependência da máquina. As inconsistentes restrições nos modos de endereçamento, sobreposição de registradores ou células de memória, efeitos colaterais de instruções e endereços, tornaram a tarefa de produzir otimizadores muito difícil, mesmo quando escritos manualmente. Para suprir estas dificuldades sentiu-se a necessidade da descrição formal da semântica do conjunto de instrução, para definir, precisamente, os conceitos básicos a nível de código de máquina, tais como os modos de endereçamento, instruções, etc. Através desta formalização as informações de contexto relevantes para as otimizações locais são definidas pela interpretação da semântica das instruções. Isto resulta em uma noção de equivalência de instruções relativas a um dado contexto. Baseado nesta noção, regras de otimizações aplicáveis em uma arquitetura de máquina específica podem ser derivadas automaticamente.

5.2 Métodos de Otimizadores "peephole" Existentes

Recentemente têm havido várias tentativas para formalizar e automatizar otimizações "peephole" , Davidson e Fraser, [DAVIDSON 80], [DAVIDSON 84a], [DAVIDSON 84b], [DAVIDSON 87], Tanenbaum et alii [TANENBAUM 82], Fraser e Wendt, [FRASER 86], Ganapathi e Fischer [GANAPATHI 88], Warfield e Bauer [WARFIELD 88], Kessler [KESSLER 84], [KESSLER 86], Massalin [MASSALIN 87] entre outros.

5.2.1 Abordagem de Davidson e Fraser (1980):

Davidson e Fraser [DAVIDSON 80] propuseram em seu trabalho um otimizador "peephole" redirecionável, denominado PO. Dado um programa em linguagem assembler e uma descrição simbólica da máquina, PO simula pares de instruções adjacentes, e, quando possível, troca-as por uma única instrução equivalente. PO faz um passo para determinar o efeito de cada instrução, um segundo passo para juntar pares com o mesmo efeito e um terceiro para selecionar a instrução mais barata para cada resultado obtido. Como resultado desta organização, PO é independente de máquina e pode ser descrito formalmente e concisamente. Quando PO termina, nenhuma instrução, ou par de instruções adjacentes, pode ser trocada por uma mais barata. Esta completeza permite ao PO isentar os geradores de código de muitas análises de casos, por exemplo, ele pode produzir somente sequências "load/add-register" e fiar-se em PO para, quando possível, descartá-los em favor de instruções "add-memory, add-immediate", ou incremento.

A arquitetura de máquina, para Davidson e Fraser é descrita através de uma gramática para tradução dirigida por sintaxe, entre a linguagem de montagem e a transferência de registradores. As produções nesta gramática são compostas de expressões e comandos simples envolvendo os registradores e células de memória da máquina alvo. A linguagem proposta provê ainda facilidades para definir não-terminais, a sintaxe em linguagem de montagem para cada não-terminal e a sintaxe para a correspondente transferência de registradores e modos de endereçamento.

Muito embora os resultados obtidos neste método sejam bons, existem duas limitações inerentes em abordagens interpretativas como esta: uma se refere ao número de instruções na janela do "peephole", e a outra é que efetuando todas as manipulações simbólicas em tempo de compilação torna PO muito lento.

5.2.2 Abordagem de Davidson e Fraser (Junho/1984):

O método proposto por Davidson e Fraser [DAVIDSON 84a], complementa a abordagem de [DAVIDSON 80]. Para melhorar o desempenho do compilador, PO é estendido para gerar regras que descrevem as otimizações que devem ser feitas. Um conjunto fixo de regras é gerado em tempo de geração de compilador e carregado em um otimizador dirigido por regras, denominado HOP. As regras em HOP são codificadas como textos, com variáveis padrões da forma % n embutidas para denotar operandos sensíveis ao contexto. HOP também usa padrões para representar instruções usando padrões. Por exemplo, a instrução "r[1] = x" é representada com o padrão "r[% 1] = % 2" acrescida da informação de que % 1 denota 1 e % 2 representa x, ou seja, quando HOP lê a instrução acima, ele imediatamente a converte ao formato padrão representando-a com a tupla "r[% 1] = % 2, 1, x". As vantagens desta representação em termos de velocidade são duas: primeiro HOP usa uma tabela "hash" para armazenar exatamente uma cópia de qualquer "string", permitindo assim, a comparação de instruções com padrões pela comparação entre dois endereços "hash"; segundo, HOP armazena regras em outra tabela "hash" chaveada pelo padrão no qual a regra se aplica.

5.2.3 Abordagem de Davidson e Fraser (1984):

Davidson e Fraser [DAVIDSON 84b] descrevem a implementação do compilador "YC" para a linguagem de programação Y, proposta por D. R. Hanson, que otimiza após a geração de código. Este trabalho também é uma extensão de [DAVIDSON 80]. Sua técnica usa descrição de instruções. O redirecionamento do otimizador para outra máquina envolve a troca da descrição das instruções. Além disto, sua técnica requer um conjunto de "programas de treinamento" que ensina ao otimizador o que pode ser otimizado. Este conjunto de programas de treinamento deve ser reescrito para cada nova arquitetura. O otimizador de código objeto é independente de máquina e geral. Ele extrai as dependências de máquina da descrição da mesma e implementa somente duas otimizações

gerais: eliminação de sub-expressões comuns e substituição de instruções adjacentes por uma única instrução equivalente. O otimizador permite o uso de geradores de código simples tornando o compilador fácil de ser redirecionado.

O compilador chamado "YC" traduz a linguagem de programação Y, proposta por D. R. Hanson, entretanto suas técnicas podem ser aplicadas a outras linguagens tipo Algol. De fato, sua estratégia de geração e otimização de código tem sido usada para construir compiladores para sub-conjuntos da linguagem Módula-II proposta por N. Wirth.

YC é organizado em cinco fases:

1. Front-end

   Compila o código fonte para o código de uma máquina abstrata.

2. Code-expander

   Esta fase expande o código da máquina abstrata em transferência de registradores, uma representação aproximadamente equivalente ao código assembler.

3. Cacher

   Elimina sub-expressões comuns nas transferências de registradores.

4. Combiner

   Faz a troca de sequências de instruções de transferência de registradores por uma única instrução equivalente.

5. Assigner

   Esta fase os traduz para código assembler.

As fases, Cacher, Combiner e Assign estendem o otimizador PO de [DAVIDSON 80].

YC usa o código de máquina abstrata para manter o "front-end" independente de máquina e usa a transferência de registradores como uma representação independente de máquina para as instruções específicas da máquina, ou seja, ela descreve o efeito das instruções de máquina. YC é facilmente redirecionável. YC roda no VAX-11 e PDP-11 sob o sistema operacional UNIX e produz código para o VAX-11, PDP-11, DECsystem-10, CDC Cyber, Motorola 68000, IBM 370, e Intel 8080. O seu redirecionamento é obtido escrevendo uma descrição da máquina, codificando um simples gerador de código, fazendo algumas trocas simples no Cacher e Assigner.

5.2.4 Abordagem de Robert R. Kessler:

Robert R. Kessler [KESSLER 84], em contraste aos trabalhos de Davidson e Fraser, [DAVIDSON 80], [DAVIDSON 84a], [DAVIDSON 84b], descreve um sistema, denomi-

nado PEEP, que ao invés de analisar as sequências de instruções que ocorrem durante a geração de código, analisa a descrição da máquina durante a construção do compilador. A técnica proposta por ele limita-se a descobrir instruções que sejam equivalentes a sequências de instruções de comprimento dois. Ao invés de usar um conjunto de programas de treinamento, ele usa a descrição da máquina para encontrar todas as otimizações possíveis. Os efeitos de um par de instrução são combinados e a descrição de instruções é pesquisada para descobrir uma única, mais eficiente e que tenha o mesmo efeito que as duas instruções combinadas. Seu sistema funciona usando o raciocínio para-frente.

Robert Kessler utiliza uma linguagem baseada em LISP que permite uma grande flexibilidade na escrita das definições. O usuário pode definir: constantes, registradores, modos de endereçamento e instruções. As instruções são descritas provendo o seu formato de entrada, as equações semânticas definindo suas funções e o seu custo englobando tempo e espaço. Na maioria das arquiteturas, cada instrução permite vários modos de endereçamento para cada operando. A linguagem proposta por Kessler permite a definição de uma enumeração na especificação de cada um dos operandos, ou seja, os operandos das instruções são definidos como um conjunto de todos os modos de endereçamento possível.

Kessler usa a seguinte técnica para combinar instruções:

1. Combine todos os pares possíveis de instruções da descrição da máquina. Se a segunda instrução referenciar um recurso definido na primeira instrução, substitua o fonte da primeira instrução em referência na segunda instrução e então pesquise por uma instrução que execute a combinação semântica. Um casamento deve ter seu custo de espaço e tempo menor que das duas instruções originais. Acrescente as três instruções na tabela.

2. Se a primeira instrução não definir um recurso usado na segunda instrução, concatene as duas instruções e pesquise por uma instrução que execute ambas instruções.

5.2.5 Abordagem de Peter B. Kessler:

Para aliviar alguns dos problemas inerentes a um modelo dirigido puramente pela sintaxe, Peter B. Kessler, [KESSLER 86] sugere a inclusão de uma fase separada de transformação de código. Assim, construções de propósitos especiais são completamente removidas da descrição da máquina. E ao invés de usar a composição "força bruta" para formar sequências de instruções, ele sugere a descoberta de idiotismos ou idiomatismos (idioms) pela decomposição. Uma sequência de instrução complexa é decomposta em uma sequência de instruções simples e por este meio determina-se sequências de código ineficientes que podem ser substituídas por outras mais eficientes. Ou seja, esta técnica identifica restrições semânticas em uma sequência de instruções de tamanho arbitrário que a tornam equivalente a uma única instrução de propósito especial, denominada idiomatismo. A decomposição não é limitada a descobrir pares de instruções equivalentes;

ela pode descobrir que uma instrução é equivalente à uma longa sequência de instruções. No pior caso, decomposição toma mais tempo que a composição, entretanto, na média, decomposição pode gastar menos tempo. Kessler identifica três classes gerais de idiomatismos: idiomatismo de conjunto (set idioms), idiomatismos de ligação (binding idioms) e idiomatismos de composição (composite idioms).

Um idiomatismo de conjunto é uma instrução de propósito especial que pode ser usada na substituição por uma instrução de propósito geral quando, um ou mais de seus operandos possuem um valor pertencente a um conjunto particular de uma máquina específica. Por exemplo, uma instrução "INC 1" pode ser um idiomatismo de conjunto para uma instrução geral "ADD", quando um dos operandos da adição tem valor um.

Um idiomatismo de ligação refere-se a uma instrução de propósito especial que executa a mesma operação que muitas instruções de propósitos gerais quando dois ou mais operandos de uma instrução geral fazem referência a mesma porção de memória.

Um idiomatismo de composição é uma instrução que executa as mesmas computações que uma sequência de instruções. Por exemplo, muitas arquiteturas possuem instruções para serem usadas ao final de "loops". Tal instrução deve adicionar um valor ao índice, compará-lo com um valor limite e desviar conforme o resultado da comparação. A instrução de "loop" de propósito especial é preferida em relação a separar as instruções de adição, comparação e desvio.

Frequentemente os três tipos de restrições semânticas associados a estes idiomatismos devem ser considerados juntos para descobrir sequências equivalentes de código em uma arquitetura alvo. No exemplo anterior, a instrução que controla o "loop" deve somente incrementar o valor do índice de 1, ao invés de permitir adições sem restrições. Ademais, o mesmo operando deve ser testado como foi incrementado para que as sequências sejam equivalentes.

A decomposição da descrição de instruções é feita da sequinte maneira: dada qualquer instrução identifica-se todas as outras sequências de código que podem ser substituídas por aquela instrução. Este processo é repetido para cada instrução da máquina alvo, produzindo uma lista de todas as restrições de equivalência. A identificação dos idiomatismos é dirigida pela classe geral de idiomatismos, mencionada acima, e pelas instruções individuais da máquina alvo. Por exemplo, se há uma instrução na máquina alvo que soma pequenas constantes, uma pesquisa será feita para outras instruções que podem ser restringidas a somar pequenas constantes.

A maior contribuição desta técnica está no fato de que sequências de instruções podem ser estendidas para comprimentos arbitrários em uma tentativa de decompor uma instrução. A complexidade do processo de análise é exponencial ao número de instruções da máquina alvo, o grau de exponenciação depende do comprimento das sequências equivalentes que foram encontradas [KESSLER 86]. Esta é uma propriedade importante, pois, quanto mais complexo for o conjunto de instruções mais tempo levará para analisar.

5.2.6 Abordagem de Davidson e Fraser (1987)

Davidson e Fraser [DAVIDSON 87] descrevem um otimizador "peephole" dirigido por regras, onde as regras são inferidas automaticamente pela execução de um conjunto de programas de treinamento através de um otimizador "peephole", que é dirigido pela descrição da arquitetura da máquina. A vantagem desta abordagem é que as regras são derivadas automaticamente. A desvantagem é que essencialmente deve se construir dois compiladores, um que opera usando a descrição da máquina e um outro que usa as regras. O segundo, entretanto, é construído automaticamente uma vez que o primeiro é completado. O resultado final é sem dúvida um otimizador "peephole" rápido.

5.2.7 Abordagem de Tanenbaum

No "Amsterdam Compile Kit" [TANENBAUM 83], são usados dois otimizadores "peephole" para substituir um gerador de código tradicional. O "front end" emite código para uma máquina abstrata. Esta máquina chamada EM simula uma máquina à pilha acrescida de uma série de operações especiais como incremento, decremento, etc. Melhorias são introduzidas no código EM por um otimizador "peephole" que substitui sequências de instruções ineficientes por outras mais eficientes. Este otimizador é dirigido por uma tabela de substituição de pares de padrões. O código da máquina abstrata melhorado é processado por um otimizador global, que efetua uma série de otimizações, que requer uma visão geral da estrutura do programa. Um "back-end" traduz o código da máquina abstrata otimizado para sequências de instruções da máquina alvo. Um segundo otimizador "peephole", específico para uma arquitetura, melhora o código assembler da máquina alvo. Este otimizador é similar àquele usado sobre o código EM.

5.2.8 Abordagem de Warfield e Bauer

Warfield e Bauer [WARFIELD 88] apresentam uma técnica que usa um Sistema Especialista com a missão de reconhecer que instruções podem ser otimizadas a partir da descrição da instruções da máquina alvo. Uma ferramenta denominada "Meta-level Representation System" (MRS) é usada na construção do sistema especialista, sendo sua característica principal a inclusão de um esquema de controle flexível utilizando raciocínio para-frente, raciocínio para trás e uma técnica denominada resolução [RICH 83] para provar teoremas, além da habilidade de representar o conhecimento sobre ele mesmo (metalevel knowledge). MRS descreve uma teoria como um conjunto de fatos em sua própria biblioteca.

Neste sistema são definidas três teorias: a teoria de descrição de instruções que descreve o que cada instrução faz, em termos das proposições entendidas pelo MRS. A teoria de modos de endereçamento que descreve os modos de endereçamento e suas classes. Finalmente a teoria de otimização, nesta teoria usando o raciocínio para trás e um "peephole"

de duas instruções as regras tentam encontrar todas as otimizações possíveis, usando certos critérios. Uma otimização é considerada válida somente se as constantes referentes ao tempo e tamanho das instruções originais são, ambas, maiores do que as constantes de espaço e tempo da nova instrução. Considera-se, também, o efeito que uma instrução otimizada tem sobre o código de condição. As regras pegam todos os pares de instruções da teoria de descrição de instruções, as combinam e tentam encontrar uma única instrução, mais eficiente, para substituir a combinação das instruções originais. Além de descobrir todas as combinações de instruções, que podem ser otimizadas, o otimizador também considera desvio sobre desvio e modos de endereçamento especiais.

O otimizador de regras retorna uma lista de todas as otimizações possíveis e as condições que devem ser verdadeiras para que o otimizador funcione. Este resultado do Sistema Especialista é colocado na forma de regras e posto em uma teoria chamada Regras. Novas regras são criadas a partir daquelas que o otimizador retorna. O raciocínio para-frente é então usado para otimizar o código objeto usando estas regras. Cada linha do código objeto é lida como um fato na biblioteca após ter sido codificada no formato definido por MRS. Após duas instruções terem sido declaradas, MRS pesquisa, automaticamente, as regras para encontrar uma que case as duas instruções. Se esta regra é encontrada, MRS instala a nova instrução otimizada na biblioteca. Esta nova instrução pode ser recuperada da biblioteca para ser usada na substituição de duas instruções originais no código objeto. Após cada otimização de duas instruções serem tentadas, as duas instruções originais junto com a terceira instrução, se houver, devem ser removidas da biblioteca.

A linguagem proposta por Warfield e Bauer baseia-se na sintaxe de LISP. Ela provê, além de facilidades para descrever instruções e modos de endereçamento, e suas respectivas classes, um mecanismo para definir regras.

Para redirecionar o Sistema Especialista proposto para outra máquina, basta substituir a descrição das instruções na teoria de instruções e a descrição dos modos de endereçamento na teoria de modos de endereçamento.

O processo de combinação de instruções usado nesta técnica é similar aquele proposto por Robert R. Kessler [KESSLER 84]. Ambos são fortemente baseados em pesquisa e reconhecimento de padrão. Todavia, a pesquisa e o reconhecimento de padrão nesta abordagem são efetuados automaticamente pelo MRS, tornando o uso de um Sistema Especialista mais desejável.

5.2.9 Abordagem de Giegerich

Giegerich [GIEGERICH 83] apresenta um método para formalizar arquiteturas de máquinas visando a derivação sistemática de otimizadores, onde a exatidão da otimização seja garantida.

A descrição da máquina é analisada dentro de uma abordagem formal. A análise deriva predicados e funções que irão testar e inferir informações sobre o fluxo de dados. Estes predicados esclarece dependências de máquina, dependência de fluxo de dados e propriedades dependentes do contexto. Os predicados dependentes de máquina nas instruções e modos de endereçamento são avaliados em tempo de geração de compiladores. Os predicados dependentes de programa devem ser avaliados em tempo de geração de código. Durante a geração de código, uma instrução é comparada com outra e substituída se for vantagem. O otimizador aplica várias transformações independentes de máquina explorando informações de fluxo de dados dependentes do programa. Estas transformações cobrem várias otimizações locais, incluindo eliminação de sub-expressões comuns e eliminação de código redundante.

5.3 Integração das fases de Geração e Otimização de Código

As fases de geração de código e otimização são muitas vezes simples reconhecimento de padrões: o gerador de código casa padrões em código intermediário e o substitui por código objeto; o eliminador de sub-expressões comuns procura por expressões repetidas e as troca por referências a registradores; o otimizador local casa padrões em código objeto substituindo-o por um código mais eficiente.

As vantagens da integração do gerador e otimizador local de código são:

1. Com pacotes distintos de geradores de código e otimizadores locais, são necessários duas descrições de máquina. Em um esquema integrado de seleção de código, a necessidade de uma descrição duplicada de descrição de máquina é eliminada. Como consequência, a duplicação do esforço no reconhecimento de padrão é também eliminado, tornando o compilador mais rápido e fácil de redirecionar.

2. Em um pacote separado de otimizador local, a janela é normalmente limitada a duas ou três instruções. O aumento desta janela degenera a rapidez do otimizador de maneira inaceitável. Entretanto, pela integração das fases de geração e otimização de código, a janela (peephole window) pode ser arbitrariamente maior, sem nenhuma penalidade no desempenho. Uma janela de instrução maior permite otimizações mais complexas, tais como, o uso de instruções de fluxo de controle, que saem fora de expressões ou blocos básicos.

5.3.1 Abordagem de Fraser e Wendt

Um compilador explorando estas observações é proposto por Fraser e Wendt [FRASER 86]. Em sua abordagem, um gerador de código e um otimizador local dependente de máquina, estão coesamente integrados. Ambas as funções são efetuadas por um

único sistema baseado na reescrita de regras (rule rewriting), que casa e substitui padrões. Esta organização torna o compilador mais simples, rápido e mais capaz de produzir um bom código.

O projeto, proposto por eles, se inicia com um otimizador local dirigido por regras e o generaliza, também, para assumir as responsabilidades da geração de código. O compilador é redirecionável. As regras de geração de código são escritas manualmente, mas sua tarefa é simplificada pela ausência de análise de casos especiais. A necessidade de escrever estas regras é compensada pelo fato, de que a descrição da máquina necessária é pequena o suficiente para tornar o método descrito competitivo com os outros métodos de compiladores redirecionáveis existentes, [CATTELL 80].

Um conjunto de regras gera um código simples e outro, o qual é geralmente criado automaticamente em tempo de geração de compiladores (compiler-compiler time), otimiza este código tão logo ele seja produzido, isto é, a combinação do gerador de código com o otimizador é um sistema geral baseado na reescrita de regras (rule-based rewriting), que casa padrões e substitui novos textos por eles. Algumas regras implementam a geração de código pela substituição do código intermediário por instruções de máquina, representadas como transferência de registradores. Outras regras implementam otimizadores locais pela substituição de instruções justapostas por uma única. Ainda outras regras traduzem as transferências de registradores otimizados para código em linguagem de montagem.

5.3.2 Abordagem de Ganapathi e Fischer

Ganapathi e Fischer [GANAPATHI 88] propõem uma metodologia para otimização local que é encapsulada em um gerador de código baseado em um analisador sintático com avaliação dos atributos (attributed-parsing) [GANAPATHI 82a]. Sua abordagem aperfeiçoa a técnica de gramática de atributo na identificação de idiomas dirigidos por descrição e na aplicação de idiomas dirigidos por tabela. A informação de estado em um bloco básico é representada, explicitamente, através de atributos, e não por meio de um avaliador de atributos. Para invocar otimizações uniformemente, a partir da gramática, o mecanismo de "buffering" [GANAPATHI 82a] é substituído por produções de gramática [GANAPATHI 87]. Além disto esta notação é usada para descrever idiomas com efeitos colaterais tais como autoincremento e autodecremento. A técnica de reconhecimento de padrão é estendida para abranger floresta de árvores de tal forma que otimizações "inter-instruction" sejam efetuadas, consistentemente, pelo analisador sintático através da avaliação dos atributos. Custos são introduzidos para dirigir a geração e otimização de código com o objetivo de evitar qualquer ordem estática das produções. Valores de atributos explícitos são incluídos aos símbolos da gramática para formar produções com atributos. Eles mostram exatamente os valores semânticos que estão associados aos símbolos da gramática e onde eles são computados. A descrição da semântica da instrução é encapsulada nestes atributos semânticos que cuidadosamente armazenam todos os efeitos colaterais, tais como, resultados múltiplos e sedimentos do código de condição. Opcionalmente,

cada produção da gramática possui, a ela associada, um conjunto de símbolos predicados e símbolos de ação. Símbolos de ação encapsulam o cálculo dos valores de novos atributos e operações de máquina que tem efeito colateral. Os predicados representam mecanismos adequados que definem exatamente as circunstâncias sobre as quais uma produção pode ser aplicada. Eles são usados para verificar as restrições semânticas dos atributos quando necessário. Em muitas produções, eles também especificam restrições de arquiteturas no uso de instruções da máquina correspondente. Nesta abordagem informações contidas nos atributos são usadas para controlar o analisador. Se o predicado for verdadeiro, o casamento entre produções prossegue. Se for falso, a produção é eliminada, ou seja, não é levada em consideração.

Como na geração de código dirigida por atributo, o desenvolvimento do otimizador local é incremental e integrado à geração de código. É fácil melhorar a qualidade do código gerado sem alterar o mecanismo básico de geração de código, adicionando novas produções de gramática, predicados ou símbolos de ação.

6 CONCLUSÃO

Tem havido muita pesquisa teórica na área de geração de código. Na prática as arquiteturas das máquinas frequentemente não correspondem aos modelos formais. A abordagem interpretativa é um melhoramento em relação a geração de código "ad-hoc" porque somente "p + m" tradutores são necessários para implementar "p" linguagens em "m" arquiteturas. Mas em tais esquemas a descrição da máquina é misturada ao algoritmo de geração de código. Redirecionamento então requer a troca do gerador de código para cada máquina nova.

A abordagem dirigida por reconhecimento de padrão separa a descrição da máquina do algoritmo de geração de código, provendo assim um nível maior de portabilidade. Em tais esquemas, o reconhecimento de padrão é usado em substituição a interpretação. Fraser, Graham-Glanville, Ripken e Cattell tentaram derivar automaticamente o gerador de código a partir da descrição da máquina, muito embora seus métodos sejam muito diferentes. O sistema de Fraser, baseado em regras é ineficiente e sua portabilidade é questionável. Ripken considerou em detalhes a interação entre as diferentes fase em um compilador. Entretanto, uma implementação de seu algoritmo usando programação dinâmica pode ser muito lento.

A abordagem dirigida por tabela é um melhoramento em relação as abordagens anteriores. Cattell usa um algoritmo de pesquisa baseado em heurística para derivar sequências de código em casos onde não há casamentos entre os operadores dos gabaritos de IR e os da máquina alvo. Tal derivação automática usando axiomas não é prática para uma variedade de instruções de máquina. O esquema de Glanville é o melhor do ponto de vista prático. Porém usando seu esquema em um compilador de produção requer que muito do

trabalho de dependência de máquina seja feito por outras fases do compilador. O gerador de código de Ganapathi et alii, é uma extensão do trabalho de Graham-Granville. O uso de atributos em sua abordagem simplificou a tarefa de interface entre as partes de um compilador dependente de máquina e partes independentes de máquina. Entretanto, o uso de uma representação intermediária mais complicada devido ao uso de gramática de atributos introduziu uma complexidade maior no projeto do compilador em relação a descrição do gerador e otimizador de código.

Os geradores de código CG [GANAPATHI 82a] e Graham-Glanville [GRAHAM 78] não são tão ambiciosos como o XGEN de Fraser [FRASER 77]. A ênfase é mais na tradução de uma representação intermediária para código objeto. Ambos os esquemas encontram uma correspondência entre a representação intermediária e a máquina alvo. As regras no CG são sequências em IR [GANAPATHI 81b] com condições especificando a aplicabilidade da tradução para a máquina alvo. Em contraste, XGEN efetua uma série de transformações e então tenta o casamento. Efetivamente ele reestrutura os padrões para efetuar o casamento. Nesta conexão não há garantia a priori que uma série de transformações seja suficiente e que um casamento encontrará uma correspondência. O CG de Ganapathi pode garantir o código, muito embora isto possa requerer a ajuda de um "machine definer". Como as abordagens de Graham-Glanville e Ganapathi são menos ambiciosas, elas podem prover esta garantia.

Um número de técnicas que podem ser protótipos para futuras pesquisas em geradores automáticos de código foram mostradas neste trabalho. Estas técnicas demonstraram a praticalidade da geração automática de código como um problema de rotina na tecnologia de compiladores.

Geradores de código baseados em recentes otimizadores locais para análise de casos especiais tem sido desenvolvidos separadamente de geradores de código baseados em análise sintática ou em casamento de árvores (tree-matching). Geradores de código baseados em casamento de árvores geralmente casam uma sub-árvore correspondendo a uma instrução, emitem aquela instrução e substituem a sub-árvore com um único nodo identificando onde aquela instrução depositou seu resultado. A instrução normalmente não participa em casamentos subsequentes. Geradores de código baseados em analisadores sintáticos podem também ser vistos como efetuando uma operação de casamento similar em uma árvore linearizada. Estes dois tipos de geradores de código geralmente se beneficiam de algum otimizador local. A abordagem de Fraser [FRASER 86] vista na Seção 5.3.1 sugere uma maneira pela qual estes desenvolvimentos podem se convergir. Seu sistema de reescrita (rewriting) também é baseado em casamento de árvores, mas ele "recycle" sua saída. Ou seja, ele normalmente aplica uma regra para gerar uma instrução e então outras regras para melhorá-la. Reciclagem (recycling) ajuda o algoritmo executar otimizações locais tão bem como geração de código.

Foi mostrado também neste trabalho que o uso de um sistema baseado no conhecimento para um otimizador local diminui o trabalho necessário para construí-lo, mantê-lo e redirecioná-lo.

Ganapathi e Fischer [GANAPATHI 88] integram um otimizador local em um gerador de código existente baseado em gramática de atributos. A estrutura resultante é independente de máquina e permite uma inclusão fácil de uma grande variedade de otimizações.

Para finalizar, vale mencionar um "surveyor's forum" sobre geradores de código automático, [SURVFORUM 83], onde, Wulf, Leverett, Cattell e Fraser esclarecem alguns pontos mal interpretados por Ganapathi et alii [GANAPATHI 81a], [GANAPATHI 82b], relacionados a seus respectivos trabalhos, [WULF 80], [LEVERETT 79], [LEVERETT 80], [CATTELL 78], [CATTELL 80] e [FRASER 77].

Bibliografia

[AHO 86] AHO, A. V., SETHI, R., and ULLMAN, J. D., Compiler Principals, Techniques and Tools, Addison -Wesley Publishing Company, 1986.

[AHO 73] AHO , A. V.,and ULLMAN, J. D., The Theory of parsing, translation and compiling, Vols. 1 e 2, Prentice-Hall, Englewood Cliffs, N.J. 1973.

[AHO 76] AHO A. V. and JOHNSON , *Optimal Code Generation for Expression Trees*, JACM Vol. 23 No. 3 pp. 488-501, 1976.

[AHO 80] AHO, Alfred V., *Translator Writing Systems: Where do They Now Stand*, COMPUTER 13, August, 1980.

[AHO 79] AHO, A. V., KERNIGHAN, B. W., and WEINBERGER, P.J., *AWK - a pattern scanning and processing language,* SOFTWARE-PRACTICE and EXPERIENCE 9:4, 267-280, 1979.

[AHO 60] AHO A. V. and JOHNSON , *Optimal Code Generation for Expression Trees*, JACM Vol. 23 No. 3 pp. 488-501, 1976.

[AMMANN 77] AMMANN U., *On Code Generation in a Pascal Compiler*, SOFTWARE - PRACTICE and EXPERIENCE, Vol. 7 No. 3 pp. 391-423, June/July 1977.

[AVISSAR 85] AVISSAR, O., *SIG - Sistema de Suporte à Implementação Automática de Geradores de Código*, Tese de mestrado, UFMG , 1985.

[BELL 71] BELL, C. G. and NEWELL, A., Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971.

[BIRD 82] BIRD, P., *An Implementation of a Code Generator Specification Language for Table Driven Code Generators*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[BACKUS 59] BACKUS, J. W., *The syntax and semantics of the proposed international algebraic language of the Zurich ACM* - GAMM Conference. Proc. International Conf. on Information Processing, UNESCO (1959), 125 - 132.

[BARR 82] BARR, A. and Feigenbaum, E. A., The Handbook of Artificial Intelligence. Vol 1. Los Altos, California:Morgan Kaufman. Eds. 1982.

[BIGONHA 85] BIGONHA, M. A. S., *SIC - Sistema de Implementação de Compiladores*, Tese de Mestrado - DCC/UFMG, 1985.

[BIGONHA 81] BIGONHA, R. S., *A Denotational Semantics Implementation System*, PhD Dissertation, University of California Los Angeles, 1981.

[BUNDY 78] BUNDY, A. *Will it reach the top? Prediction in the mechanics world*, Artificial Intelligence, 10, 129-146, 1978.

[BUNDY 79] BUNDY, A. et alii, Solving Mechanics Problems using Metalevel Inference. (ED. Michie), 50-64, 1979.

[BRACHMAN 83] BRACHMAN R. J, et alii., What Are Expert Systems ?, in Building Expert Systems, editores Hayes-Roth, E. et alii, 1983.

[BROWNSTON 85] BROWNSTON, Lee, Farrell, R., Kant E., Programming Expert Systems in OPS5. Addison-Wesley Pub. Co., Inc. - 1985.

[BROSGOL 80] BROSGOL, Benjamin M., et alli, *TCOL : Revised Report on An Intermediate Representation for the Preliminary Ada Language*, Departament of Computer Science, Carnegie-Mellon University, CMU-CS-80-105, February 1980.

[BURKE 82] BURKE, M. & FISHER JR. G. A., *A Practical Method for Syntatic Error Diagnosis and Recovery*, ACM, 1982.

[CARNOTA 88] CARNOTA R. J. and Teszkiewicz A. D., Sistemas Expertos Y Representacion del Conocimiento. EBAI. Edição EBAI - 1988.

[CARTER 77] CARTER J. L, *A Case Study of a New Code Generation Technique for Compilers*, Comm. ACM, Vol. 20, No. 12, Dec. 1977, pp.914–920.

[CATTELL 78] CATTELL, R. G. G., *Formalization and Automatic Derivation of Code Generators*, PhD dissertation, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

[CATTELL 79] CATTELL, R. G. G., Newcomer, J. M., Leverett, B. W., *Code Generation in a Machine-Independent Compiler*, SIGPLAN Conference Compiler Construction, ACM, 1979.

[CATTELL 80] CATTELL, R. G. G., *Automatic Derivation of Code Generators from Machine Descriptions*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980.

[CRAWFORD 82] CRAWFORD, J., *Engineering a Production Code Generator*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[DAVIDSON 80] DAVIDSON, Jack W., & FRASER, Christopher W., *The Design and Application of a Retargetable Peephole Optimizer*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980.

[DAVIDSON 84a] DAVIDSON, Jack W., & FRASER, Christopher W., *Automatic Generation of Peephole Optimization*, Proceedings of the ACM SIGPLAN, Symposium on Compiler Construction , SIGPLAN NOTICES, Vol. 19, No. 6, June 1984.

[DAVIDSON 84b] DAVIDSON, Jack W., & FRASER, Christopher W., *Code Selection through Object Code Optimization*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984.

[DAVIDSON 87] DAVIDSON J. W. & FRASER C.. W., *Automatiic inference and fast interpretation of peephole optimization rules*, SOFTWARE-PRACTICE AND EXPERIENCE, 17, (801-812) , 1987.

[DAVIDSON 89] DAVIDSON J. W. & WHALLEY D. B., *Quick Compilers Using Peephole Optimization*, SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 19(1), (79-97), JANUARY 1989.

[FEIGENBAUM 63] FEIGENBAUM, E. A. and Feldman, J., *COMPUTER AND THOUGHT*. New York: McGraw-Hill. Eds. 1963.

[FRASER 86] FRASER, Christopher W. & WENDT, Alan L., *Integrating Code Generation and Optimization*, ACM Sigplan Notices, 1986.

[FRASER 89] FRASER, C. W., *A Language for writing Code Generators*, SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, 1989.

[FRASER 77] FRASER C. W., *Automatic generation of code generators*, PhD dissertation Comp. Science Dept. Yale University, New Haven, Conn. 1977.

[GANAPATHI 81b] GANAPATHI, M. et alli, *Linear Intermediate Representation for Portable Code Generation*, Tech. Report #435, Computer Science Departament, University of Wisconsin-Madison, September 1981.

[GANAPATHI 87] GANAPATHI, M. and FISCHER, C. N., *Semantic Attributes Integrate Code Generation and Optimization* In: Proc. 20th Hawai i International Conference on System Sciences, p. 307-317. Kailua-Kona, Hawaiii, January 1987.

[GANAPATHI 89] GANAPATHI, M. & MENDAL, G., *Issues in ADA Compiler Technology*, COMPUTER IEEE, February 1989, (52-60).

[GANAPATHI 81a] GANAPATHI, M. and FISCHER, C. N., *A Review of Automatic Code Generation Techniques*, Tech. Report #407, Computer Science Departament, University of Wisconsin-Madison, January 1981.

[GANAPATHI 82b] GANAPATHI, M., FISCHER, C. N. and HENNESSY J. L., *Retargetable Compiler Code Generation*, Computing Surveys, Vol. 14, No. 4, December 1982.

[GANAPATHI 85] GANAPATHI, M. and FISCHER, C. N.,*Affix Grammar Driven Code Generation*, ACM Transaction Programming Language and Systems, 7, 4 Oct. 1985, 560-599.

[GANAPATHI 88] GANAPATHI, M. and FISCHER, C. N., *Integrating Code Generation and Peephole Optimization*, Acta Informatica 25, 85-109 (1988).

[GANAPATHI 82a] GANAPATHI, M. and FISCHER, C. N., *Description-Driven Code Generation using Attribute Grammars*, In: Proc. 9th POPL Conference, p. 108-119, ACM, January 1982.

[GRAHAM 79] GRAHAM, S. L., HALEY, C. B., JOY, W. N.,*Practical LR Error Recovery*, SIGPLAN Notices, August 1979.

[GANZINGER 82] GANZINGER, H., GIEGERICH R., MÕNCKE U. e WILHELM R., *A Truly Generative Semantics-directed Compilers Generator*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[GRAHAM 80] GRAHAM, Susan L., *Table-Driven Code Generation*, COMPUTER 13, August 1980.

[GRAHAM 82] GRAHAM, Susan L., et alli, *An Experiment in Table Driven Code Generation*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[GRAHAM 78] GRAHAM S. L., e GLANVILLE R. S., *A new method for compiler code generation*, In Conference Record of the Annual ACM Symposium on Principles of Programming Languages (Tucson, Ariz. Jan. 23-25). ACM, New York, pp. 231–240, 1978.

[GIEGERICH 83] GIEGERICH, R., *A Formal Framework for the Derivation of Machine Specific Optimizers*, ACM Translations on Programming Languages and Systems, Vol. 5, No. 3, July 1983 (478–498).

[GRIES 71] GRIES, David, Compiler construction for digital computers, John Wiley & Son Inc., 1971. HARMON, P and King D., EXPERT SYSTEMS. John Wiley & S. 1971.

[HARMON 85] HARMON, P. & KING D., Expert Systems. John Wiley & S. 1985.

[HAYES-ROTH 83] HAYES-ROTH F., Waterman D. and Lenat D. (Eds.), Building Expert Systems. Addsion-Wesley, 1983.

[HARADVALA 84] HARADVALA, S., KNOBE, B. and RUBIN, N., *Expert Systems for High Quality Code Generation.* Proceedings of the First IEEE Conference on AI Aplications, 1984, pp. 310-313.

[JACKSON 88] JACKSON Peter. Introduction to Expert Systems. Addison-Wesley. 1988.

[JOHNSON 75] JOHNSON, S. C., *Yacc - yet another compiler compiler*, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J. 1975.

[JOHNSON 78] JOHNSON, S. C., *A portable compiler: Theory and Practice*, In Proc. 5th ACM Symp. Principles of Programming Languages (Tucson, Ariz., Jan. 23-25), ACM New York, Jan. 1978, pp. 97- 104.

[KESSLER 84] KESSLER, Peter B., *Peep - An Architectural Description Driven Peephole Optimizer*, Proceedings of the ACM SIGPLAN, Symposium on Compiler Construction , SIGPLAN NOTICES, Vol. 19, No. 6, June 1984.

[KESSLER 86] KESSLER, Peter B., *Discovering Machine-Specific Code Improvements*, ACM Sigplan Notices, 1986.

[KRUMME 82] KRUMME, David W., & ACKLEY, David H., *A Practical Method for Code Generation Based on Exhaustive Search*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[LANDWEHR 82] LANDWEHR, R., *Experience with an Automatic Code Generator Generator*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[LEVERETT 82] LEVERETT, Bruce W., *Topics in Code Generation and Register Allocation*, Carnegie-Mellon University Computer Science Technical Report 82-130, July 1982.

[LEVERETT 79] LEVERETT, Bruce W., et alli, *An Overview of the Production-Quality Compiler-Compiler Project*, Carnegie-Mellon University Computer Science Technical Report 79-105, 1979.

[LEVERETT 80] LEVERETT, Bruce W., et alli, *An Overview of the Production-Quality Compiler-Compiler Project*, COMPUTER 13, August 1980.

[LUCENA 87] LUCENA, C. J. P, Inteligencia Artificial e Engenharia de Software, PUC - RJ. 1987.

[McDERMOTT 80] McDERMOTT, J. *R1: An expert in the computer system domain*, Proceedings of AAAI-80, 269-271, 1980.

[MASSALIN 87] MASSALIN, Henry, *Superoptimizer – A Look at the Smallest Program*, ACM Sigplan Notices, 1987.

[MILLER 71] MILLER, P. L., *Automatic Creation of a Code Generator from a Machine Description*, M.I.T. Tech Report MAC TR-85, 1971.

[MINSKY 68] MINSKY, M., Semantic Information Processing. Cambridge, Mass.: MITPress, 1968.

[MOSSES 78] MOSSES, P. O., *SIS - A Compiler-Generator System Using Denotational Semantics* , DAIMI, University of Aarhus (1978), pp. 1-17.

[NAUR 63] NAUR, P. (ED) *Revised report on the algorithmic language Algol 60*, Comm. ACM 6:1, 1963, 1-17.

[NEWCOMER 75] NEWCOMER J. M., *Machine Independent Generation of Optimized Local Code*, PhD Thesis, Computer Science Departament, Carnegie Mellon University, 1975.

[NII 86] NII, H. Penny, *Blackboard Systems*, Report No. STAN-CS-86-1123, também numerado como: KSL-86-18, Departament of Computer Science, Stanford University, June 1986.

[PATTERSON 85] PATTERSON, D. A., *Reduced Instruction Set Computers*, Communications of ACM, Vol. 28 No. 1, January 1985.

[PERKINS 79] PERKINS, D. R., e SITES, R. L., *Machine independent Pascal code optimization*, In Proc. ACM SIGPLAN Symp. Compiler Construction Denver, Colo. August 6-10 - ACM SIGPLAN NOTICES 14, 8 August, 1979, 201-207.

[RICH 83] RICH, Elaine, Artificial Intelligence, McGraw-Hill Book Company, N. Y., 1983.

[RIPKEN 77] RIPKEN K., *Formale Beschreibung von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe*, Technische Univ. Muenchen, Munich, Germany, July 1977.

[RITCHIE 79] RITCHIE, D. M., *A tour through the UNIX C compiler*, AT&T Bell Laboratories, Murray Hill, N. J., 1979.

[RICHARDSON 89] RICHARDSON, S. and GANAPATHI M., *Code Optimization Across Procedures*, COMPUTER IEEE, February 1989, (42-50).

[RUSSELL 85] RUSSELL, Stuart, *The Compleat Guide to MRS*, Stanford Knowledge Systems Laboratory, Stanford University Report no. KSL-85-12, June 1985.

[SHANONN 50] SHANONN, C. E. *Automatic chess player*. Scientific American, 182, (48). 1950.

[SHORTLIFFE 76] SHORTLIFFE, E. H., Computer-based Medicalconsultations: MYCIN, American Elsevier, New York, 1976.

[SMIT 70] SMITH, D. C. *MLISP*, Stanford Artificial Intelligence Project Memo. AIM-135, Stanford Univ., Stanford, Calif., 1970.

[SNYDER 75] SNYDER A., *A Portable Compiler for the Language C*, master's thesis, M.I.T., Cambridge, Mass., May 1975.

[STELL 50] STEEL, T. B., *UNCOL - Universal Computer Oriented Language Revised*, DATAMATION, pp. 18-20.

[SURVFORUM 83] SURVEYOR'S FORUM, *Retargetable Code Generators*, Computing Surveys, Vol. 15, No. 3, September 1983.

[TANENBAUM 82] TANENBAUM, A. S., et alli, *Using Peephole Optimization on Intermediate Code*, ACM Transactions on Programming Language and Systems, Vol. 4 Number 1, January 1982 (21 - 36).

[TANENBAUM 83] TANENBAUM, A. S., et alii., *A practical tool kit for making portable compilers*, Communications of the ACM, 26, 554-660 (1983).

[YATES 88] YATES, John S., & SCHWARTZ, Robert A., *Dynamic Programming and Industrial-Strength Instruction Selection: Code Generation by Tiring, but not Exhaustive, Search*, ACM SIGPLAN Notices, Vol. 23, No. 10, 1988.

[WARFIELD 88] WARFIELD, Jay W. & BAUER, Henry R., *An Expert System for a Retargetable Peephole Optimizer*, ACM Sigplan Notices, Vol. 23, No. 10, 1988.

[WEINGART 73] WEINGART S. W., *An Efficient and Systematic Method of Compiler Code Generation*, PhD thesis, Computer Sciences Departament, Yale University, 1973.

[WICK 75] WICK, J. D., *Automatic generation of assemblers*. PhD dissertation, Computer Science Departament, Yale University, New Haven, Conn., 1975.

[WINOGRAD 72] Winograd, T., *Understanding Natural Language*, Cognitive Psychology, 1, 1–191. 1972.

[WULF 80] WULF, Wm. A., *PQCC: A Machine-Relative Compiler Technology*, Carnegie-Mellon University Computer Science Technical Report 80-144, September 1980.

[WULF 75] WULF, W., JOHNSSON, R., WEINSTOCK, C., HOBBS, S., e GESCHKE, C.: The Design of an Optimizing Compiler, American Elsevier, 1975.

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

D C C

Geradores e Otimizadores de Código

por

Mariza Andrade da Silva Bigonha

U F M G

UNIVERSIDADE FEDERAL DE MINAS GERAIS