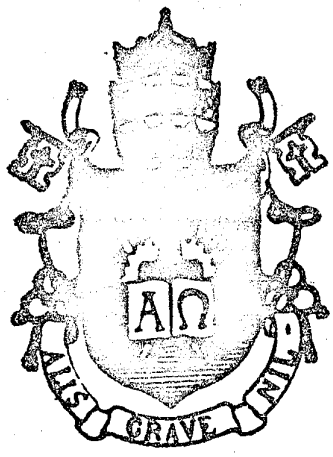

PUC

Série: Monografias em Ciência da Computação,
No. 13/90

OTIMIZACAO DE CODIGO USANDO TECNICAS DE INTELIGÊNCIA ARTIFICIAL

Mariza A. S. Bigonha

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, No. 13/90

Editor: Paulo A. S. Veloso Outubro/1990

Otimização de Código Usando Técnicas de Inteligência Artificial[1]

Mariza Andrade da Silva Bigonha

[1]Trabalho apresentado como Qualificação ao Prof. Dr. Daniel Schwabe

Parcialmente financiado pela CAPES/UFMG.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

Abstract

This technical report presents a proposal to develop a peephole optimization generator based on Artificial Intelligence techniques. In this paper, with the objective to provide a context for the problem for which we propose to investigate a solution, an overview of the methodologies for compiler construction is presented, and the role of the compiler generator systems in the development of compilers discussed. This work stresses the important aspects of the use of optimizers in the process of compilation and the current status-of-the-art with respect to most recently projected tools, giving emphasis on peephole optimization. It is also shown a general overview of the Artificial Intelligence Area and some of the current research trends in the development of peephole optimization using A.I. techniques and methodologies.

Keywords: Compilers, Code optimization, Artificial Intelligence techniques, Peephole, Compiler generator systems, Knowledge-Based Systems.

Sinopse

Este trabalho apresenta uma proposta para desenvolver um Sistema de Geração de Otimizadores Locais baseado em técnicas de Inteligência Artificial. Ao longo deste texto, com o objetivo de prover o contexto para o problema, para o qual propomos buscar uma solução, serão revistos brevemente metodologias de construção de compiladores e o papel do sistema de geração de compiladores na produção de um compilador. São mostrados a importância do uso de otimizadores no processo de compilação e o estado da arte em relação as mais recentes ferramentas criadas para o desenvolvimento dos mesmos, enfatizando os otimizadores locais. É também apresentado, uma visão geral da área de Inteligência Artificial e algumas tendências da pesquisa corrente na construção de otimizadores locais usando técnicas e metodologias de I.A.

Palavras-chave: Compiladores, Otimização de Código, Técnicas de Inteligência Artificial, Otimizadores Locais, Sistemas Geradores de Compiladores, Sistemas Baseados em Conhecimento.

SUMÁRIO

1 CARACTERIZAÇÃO DO PROBLEMA

"The most valuable of all talents is that of never using two words when one will do [in code generation]". (Thomas Jefferson)

1.1 METODOLOGIA DE CONSTRUÇÃO DE COMPILADORES

1.1.1 Introdução

Este trabalho apresenta uma proposta para desenvolver um Sistema de Geração de Otimizadores Locais baseado em técnicas de Inteligência Artificial. A motivação para esta pesquisa tem em vista a crescente proliferação de linguagens de alto nível e máquinas com arquiteturas diferentes a qual estimula a busca de métodos de implementação de compiladores que facilitem o processo de tradução e evitem a necessidade de um novo compilador para cada nova combinação linguagem fonte e máquina alvo.

Um método comum para a construção de compiladores portáteis é usar como modelo uma máquina abstrata [TANENBAUM 82], [DAVIDSON 89]. Nesta técnica, o "front-end" do compilador emite código para uma máquina abstrata, e um compilador para uma arquitetura de máquina específica deve ser projetado, construindo-se um "back-end" que traduza as operações da máquina abstrata para sequências de operações semanticamente equivalentes da máquina dada. Esta abordagem simplifica a implementação de um compilador sob vários aspectos. Primeiro, ela divide o compilador em duas unidades funcionalmente bem definidas, o "front-end" e o "back-end". Assim a análise da linguagem é formalizada e incorporada em "front-ends", da mesma forma, detalhes da máquina alvo são colocados nos "back-ends". Problemas relacionados com a arquitetura da máquina alvo são eliminados da implementação dos "front-end", e o "back-end" abstem-se de considerar aspectos particulares de cada linguagem. Além disto, projetando a máquina abstrata cuidadosamente para suportar uma única linguagem fonte, a geração de código no "front-end" é geralmente facilitada. Finalmente como o "front-end" é independente da arquitetura da máquina, um compilador para uma nova máquina pode ser obtido simplesmente provendo um novo "back-end".

Muito embora o uso de uma máquina abstrata simplifique a construção de um compilador redirecionável [AHO 86], a qualidade do código emitido é normalmente inferior ao código produzido por um compilador construído usando um gerador de código sofisticado. Por exemplo, o código produzido pelo VPCC (very portable C compiler) [DAVIDSON 89] e o "back-end" para o VAX-11 executam duas vezes mais lento que o código produzido pelo compilador C Berkely 4.3BSD. Assim, admite-se o uso de máquina abstrata para a portabilidade, desde que acrescida de um otimizador para melhorar a qualidade do código gerado.

Os melhores compiladores otimizadores existentes são baseados, na sua maioria, em algoritmos "ad hoc" e heurísticos, representando um grande investimento de tempo e esforço em sua implementação.

Há duas estratégias que podem ser seguidas na tentativa de melhorar esta situação. A primeira, que é exemplificada pelo projeto PQCC [WULF 80], é a automatização do processo de construção de compiladores. Neste caso, cria-se um programa que constrói compiladores otimizadores, de modo que, o grande investimento requerido sob a metodologia existente seja feita pelo programa ao invés de ser feita pelo programador. A outra estratégia é desenvolver uma metodologia mais eficiente para usá-la em compiladores otimizadores, de tal forma que o esforço exigido para produzir um compilador otimizador seja o menor possível. A nossa proposta se encaixa nesta estratégia.

1.1.2 Sistemas Geradores de Compiladores

Introdução: A tarefa de produzir compiladores em uma linguagem de montagem (Assembler) geralmente toma uma quantidade de tempo razoável, devido, principalmente, aos inúmeros detalhes envolvidos. A construção do compilador FORTRAN I para o IBM 704 ilustra como esta atividade pode consumir tempo. Foram necessários dezoito homens-ano para colocá-lo em funcionamento, e o esforço de dez homens-ano para estendê-lo ao FORTRAN II para o IBM 709 [BACKUS 59]. É óbvio que a grande quantidade de tempo gasto em tais projetos também reflete o estado da arte naquele tempo. Hoje, com o advento de inúmeras técnicas de compilação, o número de homens-ano necessário para efetuar a mesma tarefa foi substancialmente reduzido. Entretanto, mesmo agora, a codificação manual de um compilador para uma linguagem tipo ADA gastaria de dois a quatro homens-ano.

Em muitos casos, o tempo gasto para produzir compiladores pode ser um parâmetro muito importante. Por exemplo, a comercialização de uma nova máquina pode ser seriamente prejudicada se o compilador para ela não estiver disponível no devido tempo. Além disto, a descoberta de características mal definidas em uma linguagem podem aparecer somente após o compilador ter sido concluído, ocasião em que as correções e modificações podem ser mais caras e talvez até impraticáveis.

Felizmente, pesquisas e experiências com compiladores têm provido um melhor entendimento do processo de compilação bem como dos fundamentos teórios das técnicas envolvidas. Em particular, a observação de que compiladores efetuam tarefas em comum, como por exemplo, análise léxica e sintática, geração de código, etc., tornou possível a automatização de várias porções destas tarefas, o que vale dizer, Sistemas de Geração de Compiladores puderam ser desenvolvidos. Estes sistemas, também conhecidos como "compiler-compilers" ou "translator writing systems", variam desde sistemas muito simples, orientados a auxiliar somente em partes específicas do processo de compilação, a sistemas grandes e complexos, capazes de gerar todo o compilador a partir de uma de-

finição adequada da linguagem e da arquitetura da máquina alvo.

Particularmente com respeito à automatização da análise sintática da linguagem de programação, têm-se YACC [JOHNSON 75], SIC [?] entre outros. Menos progressos têm sido obtidos na automatização da segunda parte do processo de compilação, ou seja, na tradução da representação interna para o conjunto de instruções da máquina alvo. Acreditamos isto seja devido primeiramente à formalização inadequada das máquinas e do processo de geração de código, e não das dificuldades em automatizar o processo em si. Tentativas neste sentido tendo em vista a geração de código e a geração automática de geradores de código são encontradas hoje na literatura, [AHO 76], [AMMANN 77], [BIRD 82], [CRAWFORD 82], [GANAPATHI 81a], [GANAPATHI 89], [GRAHAM 78], [GRAHAM 80], [GRAHAM 82], [KRUMME 82], [LEVERETT 82]. Surgiram, também, os trabalhos de Cattell [CATTELL 78], [CATTELL 79], [CATTELL 80], Ganapathi et alii [GANAPATHI 81b], [GANAPATHI 82b], [GANAPATHI 85], Kessler [KESSLER 86], [LANDWEHR 82], [MILLER 71], [YATES 88], [HARADVALA 84], além de projetos importantes de geradores de compiladores, como: PQCC - Production Quality Compiler-Compiler, [WULF 80], Leverett et alii [LEVERETT 79], [LEVERETT 80], Ganzinger [GANZINGER 82], Avissar [AVISSAR 85], ECS - Experimental Compiling Systems, [CARTER 77], ACK - Amsterdam Compiler Kit, [TANENBAUM 83] entre outros.

Muito embora os Sistemas Geradores de Compiladores (SGC) possam melhorar substancialmente a rapidez com que um compilador é produzido, eles, certamente não produzem compiladores tão eficientes como aqueles projetados à mão. A principal razão é que os compiladores gerados por um SGC usam algoritmos mais gerais do que aqueles usados nos compiladores gerados manualmente. Por outro lado, os compiladores gerados através de um SGC são superiores aos projetados à mão no que diz respeito a sua integridade. Isto se deve a utilização de linguagens de propósitos especiais, como aquelas providas pelo SGC, que permitem ao projetista focalizar sua atenção em aspectos relevantes da compilação, tais como o que um compilador deve fazer, ao invés de se preocupar com minúcias inerentes em linguagens de programação de baixo nível.

Mesmo produzindo compiladores menos eficiente, o SGC irá, a longo prazo, substituir as linguagens de programação de mais baixo nível por ferramentas adequadas para implementar compiladores, assim como as linguagens de alto nível tomaram o lugar das linguagens de montagens em muitas aplicações.

Arquitetura de um Sistema Gerador de Compiladores: Superficialmente falando, um SGC é um compilador para uma linguagem orientada à escrita de compiladores. Idealmente, tal linguagem deveria permitir não só a descrição da linguagem fonte, ou seja, sua sintaxe e semântica, como aquela da linguagem objeto. Desta maneira, o SGC pode estabelecer uma ligação entre as linguagens especificadas produzindo um compilador apropriado.

Infelizmente, um sistema tão geral como este ainda não pode ser implementado. A questão

de como o relacionamento entre a semântica de duas linguagens pode ser automaticamente estabelecido é difícil de ser respondido, e continua sendo um problema em aberto. Até agora, é da responsabilidade do projetista prover este relacionamento, explicitamente, por meio das chamadas rotinas semânticas [AHO 86].

Do ponto de vista estrutural, todos os SGC são similares. Eles consistem basicamente em dois grandes componentes, o gerador de analisador sintático e o gerador de código. O gerador de analisador sintático tem por objetivo produzir o algoritmo de análise sintática do compilador a partir da definição da sintaxe da linguagem, e compete ao gerador de código produzir o módulo de geração de código do compilador.

Geralmente o gerador do analisador sintático produz tabelas a partir da gramática da linguagem para a qual o compilador está sendo escrito. Estas tabelas são usadas pelo algoritmo básico do analisador sintático, o qual é fixo para todos os compiladores eventualmente gerados pelo sistema. As gramáticas livres-do-contexto, em particular a BNF (Backus Naur Form), 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 pode-se construir automaticamente um analisador sintático eficiente [AHO 86], [BIGONHA 85].

O outro importante componente de um SGC é o gerador de código; ele é normalmente especificado por um conjunto de rotinas semânticas. Tradicionalmente, estas rotinas semânticas são escritas em uma linguagem procedural, e quando executadas traduzem programas da linguagem fonte para uma outra forma equivalente. Do ponto de vista de um compilador, este conjunto de rotinas forma uma espécie de biblioteca, para a qual o compilador pode extrair aquelas necessárias para uma compilação em particular [GRIES 71].

Em sistemas mais modernos, tais como "SIS" (Semantics Implementation System), proposto por Peter Mosses em [MOSSES 78] e estendido em [BIGONHA 81], estas rotinas semânticas foram substituídas por uma descrição semântica da linguagem fonte, usando métodos formais para definição semântica de linguagem de programação. Em SIS, por exemplo, ao invés de um conjunto de rotinas semânticas a definição denotacional da linguagem é fornecida. Basicamente esta definição consiste em um sistema recursivo de equações, as quais são definidas em um estilo orientado à sintaxe abstrata [MOSSES 78].

Uma compilação, em um sistema como SIS, consiste em selecionar todas as equações que são relevantes ao programa que está sendo compilado, e então resolvê-las. O resultado é normalmente, mas não necessariamente, uma função a qual fornece o significado do programa. Esta função representa o código para o programa, e em SIS ela é codificada como uma variação da notação de λ-calculus [MOSSES 78].

Como se vê, a especificação de semântica de linguagens é mais complexa, sendo que, vários métodos foram propostos, e progresso considerável tem sido obtido nesta área. En-

tretanto, até hoje não há especificação formal de semântica através da qual compiladores de produção de qualidade possam ser gerados automaticamente. O trabalho de Peter Mosses, descrito acima, permitiu a geração automática de um interpretador a partir da especificação da semântica denotacional de linguagens. Todavia estudos adicionais ainda são necessários para possibilitarem a tradução da especificação da semântica denotacional de uma linguagem em um gerador de código eficiente.

1.2 OTIMIZADORES

1.2.1 Introdução

A otimização de código tem sido uma área de pesquisa por quase três décadas. Entretanto, devido às diferenças semânticas entre as linguagens, fonte e objeto, e o desconhecimento dos fundamentos teóricos naquela época, os 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 usar esta informação durante a otimização. Contudo, estes desenvolvimentos só começaram a ser, efetivamente, incorporados à compiladores em meados de 1970 [WULF 75], e a 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], 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, senão ela é denominada global. Normalmente as transformações a nível local são executadas em primeiro lugar.

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 reconhecedoras 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. Este tópico será abordado na seção 1.2.3.

O código no "peephole" não necessita ser contíguo, muito embora algumas implementações o exijam. É uma característica deste método que cada melhoria obtida é uma oportunidade para outros melhoramentos. Em geral, são necessárias 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, superposição 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, surgiu 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.

1.2.2 A Necessidade de Otimizadores

Na década passada, a otimização de código se tornou importante, e, portanto, necessária no processo de compilação principalmente devido ao tempo crítico das aplicações existentes. Em particular, no universo da linguagem Ada [RICHARDSON 89], para os sistemas portadores de aplicações, onde tempo e espaço são críticos, é indispensável o uso de compiladores otimizantes. Ademais, o desempenho de novas tecnologias de arquiteturas, tais como arquiteturas RISC (reduced-instruction-set) [PATTERSON 85] é fundamentalmente dependente de compiladores otimizantes.

A estratégia de geração de código baseada em comando por comando frequentemente produz um código objeto repleto de instruções redundantes e construções pobres. A qualidade de tal código pode ser melhorado aplicando transformações no programa objeto.

Existem inúmeras maneiras pelas quais um compilador pode melhorar o desempenho de um programa sem alterar a função computada pelo mesmo. Por exemplo, através de transformações local e global, tais como eliminação de sub-expressão comum, propagação de cópia, eliminação de código morto e propagação de constantes [AHO 86]. Estas transformações reduzem significantemente o tempo e espaço gastos.

O código de máquina produzido por um tradutor de linguagens de alto nível não é tão compacto e eficiente como poderia ser se produzido por um competente programador de linguagem de montagem. Contudo, o impacto de várias ineficiências pode ser moderado através de várias técnicas de otimizações. Um tipo de ineficiência, que é pouco tratada pelas técnicas existentes, diz respeito à presença de uma quantidade desnecessária de desvios incondicionais ou "jumps" no código objeto. A ordem linear em que o código é gerado tende a ser, precisamente, a ordem em que os comandos no código fonte são fornecidas ao compilador pelo programador, consequentemente, a transferência de controle entre blocos de código de máquina não-contíguos é obtida através de desvios muitas vezes desnecessários. Muito embora a ordem do código fonte possa ser defendida em favor da legibilidade, estética, tradição, ou sintaxe da linguagem, não há razão para que esta ordem seja preservada no código objeto.

Um outro ponto suceptível a otimização são os "loops", especialmente os "loops" mais internos, onde programas tendem a gastar a maior parte de seu tempo. O tempo de execução de um programa pode ser melhorado reduzindo-se o número de instruções nos "loops" mais internos, mesmo que esta redução acarrete um acréscimo de código fora daquele "loop". Três técnicas são importantes na otimização dos "loops":

1. Movimentação de código
   Responsável pela movimentação de expressões para fora de "loops" quando os valores por elas computadas não dependem de variáveis dentro do mesmo.

2. Eliminação de variáveis de indução
   Uma variável x é denominada uma variável de indução de um "loop" L se toda vez

que a variável x mudar de valor, ela é incrementada ou decrementada por alguma constante.

3. Redução do Custo de Operação
   Responsável pela conversão de expressões da forma "K * loop-index" em progressões aritméticas equivalentes "TMP = TMP + K".

Frequentemente a arquitetura da máquina alvo contém instruções de propósito especial que executam as mesmas operações que instruções de propósito mais geral, entretanto em um contexto mais restrito. Outras vezes, as arquiteturas possuem instruções de propósito especial equivalentes a sequências de instruções mais gerais. As instruções de propósito especial são normalmente menores, ou mais rápidas, que uma instrução de propósito geral ou uma sequência de instruções equivalente. A qualidade do código gerado é, portanto, altamente dependente do conhecimento das características especiais da arquitetura da máquina alvo. Por exemplo, há um ganho substancial no uso da instrução "BXLE" do IBM/370, a qual combina um "ADD", "COMPARE" e "BRANCH". O uso destas instruções exóticas no gerador de código é essencial para obter um alto nível de desempenho.

1.2.3 Propriedades e Aplicações de Otimizadores Locais

As melhores transformações efetuadas em um programa são aquelas que proporcionam os melhores resultados envolvendo um esforço mínimo. Antes de apresentar os diferentes métodos de otimizações "peephole" implementados, é importante rever as várias propriedades que devem ser mantidas durante as transformações ocorridas nos programas. As transformações providas por um compilador otimizante devem possuir algumas propriedades: em primeiro lugar, uma transformação deve preservar o significado do programa, ou seja, uma otimização não deve modificar a saída produzida por um programa para uma determinada entrada ou ocasionar um erro, tal como uma divisão por zero que não havia na versão original do programa fonte; em segundo lugar, uma transformação deve, em média, acelerar programas em uma quantidade razoável. Muito embora, o tempo de CPU não seja mais um fator dominante como foi antes, ainda existe situações em que uma execução rápida ou um código compacto são necessários. Finalmente a transformação deve ser lucrativa.

Como consequência da segunda propriedade, "uma transformação deve, em média, acelerar programas em uma quantidade razoável", a questão de onde realizar a otimização é levantada. Existem três possibilidades: no "front-end", no código intermediário e no "back-end". Se a primeira possibilidade for escolhida, muitas otimizações comuns terão que ser programadas para cada "front-end", aumentando assim o esforço empregado em seu desenvolvimento. Se a otimização for efetuada sobre o código intermediário, ela será implementada uma única vez e permanecerá válida para todos os "front-ends" e "back-ends. Finalmente, com a colocação de otimizações no "back- end", muito embora acarrete

em uma duplicação de esforços devido ao fato de que as mesmas terão que ser programadas para cada "back-end", os resultados obtidos serão superiores.

Tanenbaum et alii [TANENBAUM 82] discutem a realização de otimizações no código intermediário. Segundo eles, muito embora não seja possível executar todas as otimizações no código intermediário, está claro que todas as otimizações que possam ser efetuadas sobre o mesmo devem ser realizadas, a não ser pelo grau de complexidade ou pelo custo nelas envolvidas. Completando suas idéias, sabe-se que vários tipos de otimizações só podem ser introduzidas, ou são mais facilmente obtidas, se efetuadas sobre o código intermediário. Por exemplo, suponha que tenhamos que decidir se as instruções do PDP-11 "ADD # 1,R0" e "INC R0" são equivalentes ou não. Sabemos que a instrução "ADD # 1,R0" liga o "bit" de código de condição (carry) e a instrução "INC R0" não o faz, portanto não são totalmente equivalentes. Para obter esta conclusão o otimizador necessita efetuar algumas análises "live/dead" [AHO 86] para cada ocorrência de ADD # 1,RO, e verificar se o "bit" de vai-um será usado posteriormente. Como o código intermediário pode ser projetado para se abster destas idiosincrasias, otimizações podem ser realizadas sem a necessidade da informação de contexto "live/dead". O mesmo não ocorre se tal otimização for realizada no código objeto. Por outro lado, há certas otimizações que não são possíveis serem realizadas no código intermediário, pois são otimizações inerentemente dependentes de máquina, como por exemplo, a manipulação de modos de endereçamento e a determinação de quando usar desvios longos ou curtos.

A conclusão que se tira quanto ao local onde otimizar é que mesmo otimizando-se o código na forma intermediária isto não elimina a possibilidade de otimizações no código objeto.

1.2.4 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.

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 formal e concisamente. Quando PO termina, nenhuma instrução, ou par de instruções adjacentes, pode ser trocada por uma mais barata. Esta

As regras em HOP são automaticamente inferidas através do comportamento de PO, ou seja, quando PO efetua a substituição acima, ele monta uma cópia textual da troca

```
r[2] = m[x]
r[2] = r[2] - m[y]
=
r[2] = m[x] - m[y]
```

as regras em HOP são formadas parametrizando automaticamente tais cópias. A parametrização substitui cada constante distinta por uma variável de padrão diferente. Assim, o texto acima é transformado em

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
=
r[$1] = m[$2] - m[$3]
```

o qual é a regra mostrada no início desta seção.

PO armazena também o último uso de cada registrador, pois isto lhe permite evitar efetuar transformações que poderiam modificar o efeito do programa. Quando esta informação é usada, ela é anotada na cópia para que seja automaticamente parametrizada como o restante das regras. Por exemplo,

```
r[2] = i
r[3] = m[r[2]] (r[2] dead)
=
r[3] = m[i]
```

produz a regra

```
r[$1] = $2
r[$3] = m[r[$1]] (r[$1] dead)
=
r[$3] = m[$2]
```

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 manipulaçoes simbólicas em tempo de compilação torna PO muito lento.

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 (compiler-compiler time)e carregado em um otimizador dirigido por regras, denominado HOP. As regras em HOP são codificadas como textos, com variáveis de padrão da forma $ i para denotar operandos sensíveis ao contexto. HOP também usa padrões para representar instruções usando padrões. A regra

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
=
r[$1] = m[$2] - m[$3]
```

especifica que uma transferência entre registradores como

```
r[2] = m[x]
r[2] = r[2] - m[y]
```

deve ser substituída por

```
r[2] = m[x] - m[y]
```

O exemplo abaixo ilustra a otimização efetuada no código VAX para "j = j + 4"

Código intermediário na forma posfixada	Código objeto
1. push i	r[2] = m[i]
2. pushc 4	r[3] = 4
3. add	r[2] = r[2] + r[3] (r[3] dead)
4. pop j	m[j] = r[2] (r[2] dead)

Inicialmente a regra

```
r[$1] = $2
r[$3] = r[$3] + r[$1] (r[$1] dead)
=
r[$3] = r[$3] + $2
```

substitui as instruções 2 e 3 por

```
r[2] = r[2] + 4
```

em seguida, a regra

```
r[$1] = m[$2]
r[$1] = r[$1] + $3
=
r[$1] = m[$2] + $3
```

combina a instrução 1 com esta nova instrução produzindo

```
r[2] = m[i] + 4
```

Finalmente a regra

```
r[$1] = m[$2] + $3
m[$4] = r[$1]  (r[$1] dead)
=
m[$4] = m[$2] + $3
```

substitui esta última instrução e a instrução 4 por

```
m[j] = m[i] + 4
```

a qual representa a instrução do VAX

```
addl3 $4,i,j
```

Concluindo, as quatro instruções originais foram substituídas por uma única.

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.

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.

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, denominado 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 [RICH 83] da Inteligência Artificial.

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.

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 "força bruta" para formar sequências de instruções, ele sugere a descoberta de idiotismos ou idiomatismos (idioms) pela decomposição. Uma sequência de instruções 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 tornem 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 às 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 será gasto na análise.

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.

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.

Abordagem de Warfield e Bauer: Warfield e Bauer [WARFIELD 88] apresentam uma técnica que usa um Sistema Especialista [RICH 83] com a missão de reconhecer se 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 consideram 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.

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 esclarecem 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 vantajoso. 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.

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.

Abordagem de Fraser e Wendt: Fraser e Wendt [FRASER 86] propõem que um gerador de código e um otimizador local dependente de máquina, estejam coesamente integrados, e as suas funções sejam 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.

1.3 INTELIGÊNCIA ARTIFICIAL E COMPILADORES

1.3.1 Histórico

Inteligência Artificial, segundo Barr e Feigenbaum [FEIGENBAUM 63], é uma parte da Ciência da Computação que diz respeito ao desenvolvimento de sistemas inteligentes, ou seja, sistemas que produzem resultados normalmente associados à inteligência humana. Entretanto, esta definição é muito geral, pois ela engloba todas as atividades vinculadas à Ciência da Computação, uma vez que é necessário o uso da inteligência para realizar operações simples como uma soma e/ou para calcular o coseno de um ângulo. Poderiamos, então, dizer que todo sistema de computação é, sem dúvida, um sistema inteligente.

Denominaremos Sistemas Convencionais aqueles sistemas que executam funções da inteligência humana que sabemos modelar, e Sistemas Inteligentes aqueles que executam funções que, até o momento, estamos aprendendo a modelar e que só recentemente temos aprendido a fazê-lo. Neste sentido, os Sistemas Baseados em Conhecimento são inteligentes.

A Inteligência Artificial pode ser classificada em três áreas de pesquisa relativamente independentes: Processamento de Linguagem Natural, Robótica e Sistemas Baseados em Conhecimento. A primeira área de pesquisa se ocupa do desenvolvimento de sistemas que podem ler, falar ou entender linguagens naturais. A segunda área está preocupado com o desenvolvimento de robots inteligentes. E o terceiro ramo de pesquisa em Inteligência Artificial se ocupa do desenvolvimento de sistemas que usam o conhecimento simbólico para simular o comportamento do especialista. É neste terceiro ramo da Inteligência Artificial que estamos interessados.

Segundo Jackson [JACKSON 88], a área de Inteligência Artificial, como um todo, desde o

seu surgimento até hoje, passou por três períodos distintos, o que ele chamou de período clássico, período romântico e período moderno.

O período clássico teve início com a publicação de Shannon [SHANONN 50] sobre jogo de xadrez e terminou com a publicação de Feigenbaum e Feldman [FEIGENBAUM 63].

O período romântico teve início em meados de 1960 e terminou em meados de 1970. Neste período a atenção dos pesquisadores estava voltada para o problema de processamento de linguagens naturais, especialmente estórias e diálogos. O trabalho de [MINSKY 68] contém uma coletânea de trabalhos da primeira metade desta época, entretanto, vamos encontrar em [WINOGRAD 72], no sistema SHRDLU, o clímax deste período. Seu sistema consiste em um programa capaz de entender um sub-conjunto do Inglês, pela representação e argumentação sobre um domínio bem restrito, ou seja, um mundo consistindo de blocos de brinquedo.

Nota-se, que durante estes dois períodos, os pesquisadores procuravam simular o complicado processo do pensamento, mediante a procura de métodos gerais para solucionar uma classe muito ampla de problemas. Isto resultou em programas de propósitos gerais, produzindo alguns resultados parciais, mas não tão significativos como se esperava. Quanto mais ampla era a classe de problemas que um programa podia solucionar, mais pobre era o resultado de cada problema em particular.

O período moderno começou em 1975 e se estende até hoje e é caracterizado por uma autoavaliação. O casamento com os aspectos psicológicos do entendimento é menos evidente que nos anteriores. Na tentativa de fornecer um novo enfoque ao probema dos períodos anteriores a pesquisa concentra-se na procura de métodos e técnicas gerais para utilizá-los em programas mais especializados. Assim, esforços concentram-se em aspectos como o da representação e da busca. Representação, diz respeito a formulação de um problema de tal forma, que seja fácil solucioná-lo. E busca implica no método adotado para controlar de maneira inteligente a busca da solução do problema de modo que não haja um consumo excessivo de recursos, como tempo e capacidade de memória.

Até o final de 1970, novos avanços foram registrados, porém os resultados ainda eram insatisfatórios. Nesta época, descobriu-se que o poder de um programa para resolver um problema é proveniente, essencialmente, do conhecimento que o mesmo possui e não de formalismos particulares e esquemas de inferência que ele emprega. Um programa é inteligente, na medida em que o mesmo possui uma grande quantidade de conhecimento específico e de boa qualidade com respeito ao problema que se pretende solucionar.

A partir de então, a ênfase no campo teórico concentrou-se nos problemas relacionados ao conhecimento, sua aquisição, sua representação, sua conservação e sua utilização. No que diz respeito às aplicações práticas, começaram a ser desenvolvidos programas de propósitos específicos, sistemas possuindo muito conhecimento acerca de uma área limitada de problemas. Em muitos casos, estes programas foram denominados Sistemas Baseados em Conhecimento ou Sistemas Especialistas. Dentre os sistemas desenvolvidos temos: MY-

CIN, R1, MECHO,etc. MYCIN foi desenvolvido na Universidade de Stanford em meados de 1970. Ele foi projetado para auxiliar os médicos no diagnóstico e tratamento de meningites e infecções bacteriológicas [SHORTLIFFE 76]; R1 é um programa que configura o sistema VAX [McDERMOTT 80]; e MECHO é um programa cujo objetivo é resolver um grande número de problemas em Mecânica Newtoniana ou Clássica [BUNDY 78], [BUNDY 79]. Nestes sistemas um número de princípios surgiram, os quais os distinguem da programação convencional e dos mais recentes trabalhos em Inteligência Artificial.

1.3.2 Sistemas Baseados em Conhecimento

Percebe-se através da literatura, [BRACHMAN 83], [BROWNSTON 85], [CARNOTA 88], [HARMON 85], [LUCENA 87], [JACKSON 88], que não é possível definir com exatidão um conceito tão dinâmico como o de Sistema Baseado em Conhecimento (SBC), entretanto, uma definição mais abrangente é:

"Um sistema baseado em conhecimento é um sistema de computação inteligente baseado no conhecimento adquirido de um especialista, na resolução de um problema significativo em um domínio específico" [CARNOTA 88].

Define-se especialista, o indivíduo que possui um alto grau de conhecimento que lhe permite resolver certos problemas de uma maneira mais eficaz que a maioria das outras pessoas. Este conhecimento geralmente é de dois tipos, público e privado. O público, também denominado "fatos" inclui as definições e teorias que normalmente encontra-se nos livros e manuais. O conhecimento privado, também, denominado Heurística, consiste basicamente em regras práticas, resultado de vários anos de experiência. São justamente estas heurísticas, que permitem ao especialista resolver os problemas, reconhecer rapidamente qual é o enfoque mais apropriado para atacá-lo, manipular efetivamente com dados incompletos, imprecisos, etc.. Entender, adquirir e reproduzir este tipo de conhecimento é a tarefa central na construção de um sistema baseado em conhecimento.

Um sistema baseado no conhecimento distingue-se dos demais programas em Inteligência Artificial sob vários aspectos, os mais importantes são:

1. Ele trata de problemas realmente complexos, portanto envolve muitos especialistas. Ressalte-se que, nem todo sistema complexo em Inteligência Artificial baseado em conhecimento é um Sistema Especialista. Os sistemas baseados em conhecimento são usados com objetivos muito bem definidos como: diagnóstico, monitoramento, análise, interpretação, consulta, planejamento, projeto, explicação, aprendizado e conceituação.
2. Ele deve ser veloz o suficiente para ser uma ferramenta útil;
3. e ele deve ser capaz de explicar e justificar as soluções e recomendações propostas, de tal maneira que convença o usuário que seus argumentos estão de fato corretos.

Um sistema baseado em conhecimento é normalmente formado por uma coleção de proposições e regras. As proposições são fatos descrevendo os objetos que representam a área do conhecimento. As regras usam as proposições para inferir novas proposições. As regras por sua vez, são formadas por uma parte chamada condição e uma outra denominada ação. "Ação" é inferida, somente se, a "condição" for verdadeira. As regras podem usar o raciocínio para frente e para trás. A Figura 1 apresenta uma visão geral dos dois tipos de raciocínio ou encadeamento. Sucintamente, o raciocínio para trás começa na "ação" de uma regra e tenta satisfazer a "condição". O raciocínio para frente se inicia com a "condição" de uma regra e infere a "ação" se a "condição" for satisfeita.O raciocínio para trás é útil na prova de um objetivo em particular, e o raciocínio para frente é útil na busca do que pode ser inferido a partir de uma proposição. Se há várias regras diferentes para um determinado objetivo, o raciocínio para trás deve tentar todas estas regras, portanto, neste caso o raciocínio para frente é mais eficiente. Mas, se um determinado fato aparece na "condição" de várias regras, o raciocínio para trás é mais eficiente, pois o raciocínio para frente deveria tentar todas estas regras.

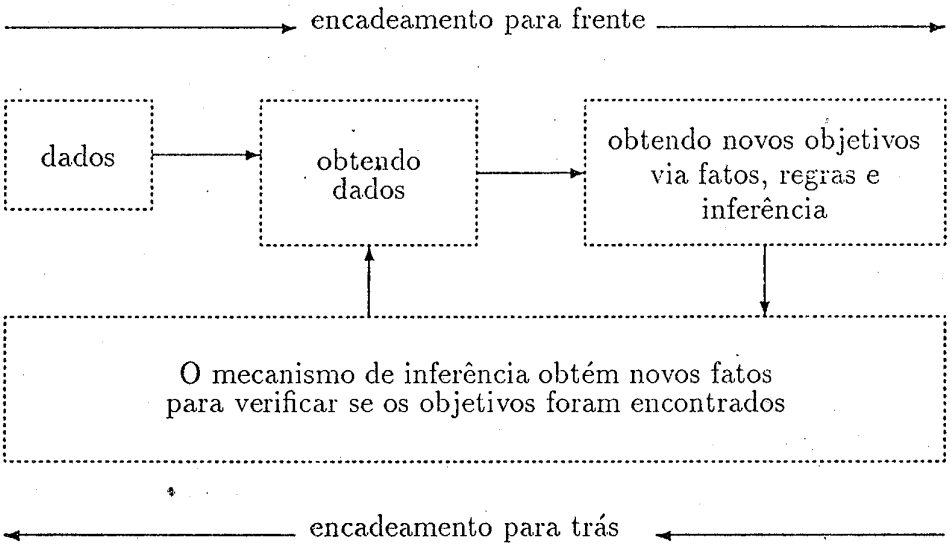

Figura 1: Visão geral dos encadeamentos para-frente e para-trás

Idealmente um sistema especialista deveria ser composto de um processador de linguagens, de uma memória de trabalho, de uma base de conhecimento, de um interpretador, de um escalonador, de um módulo de consistência e de um módulo de explicação [HAYES-ROTH 83] conforme ilustrado na Figura 2. Todavia, não se tem notícias de um sistema contendo todos estes componentes. A maioria deles contém um, ou alguns destes componentes.

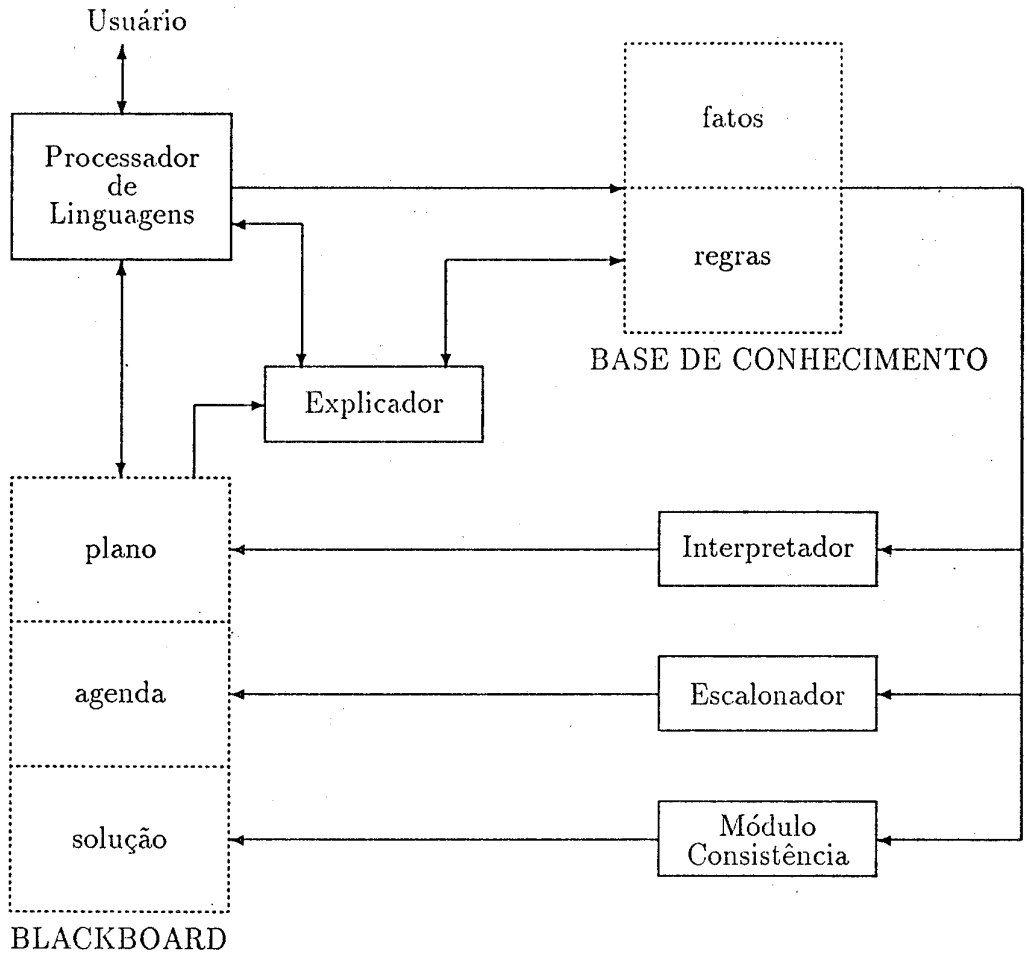

Figura 2: Componentes de um Sistema Especialista

O Processador de Linguagens é usado na interação do usuário com o sistema, usualmente esta interação é feita via uma variação da linguagem natural, ou via editores gráficos ou de texto.

A Memória de trabalho é responsável pelo armazenamento de resultados intermediários e decisões manipuladas pelo sistema especialista. Todo sistema especialista usa algum tipo de representação para os resultados intermediários. Os tipos de decisões podem ser: plano, agenda e soluções. Plano representa o método de ação usado para solucionar o problema, a agenda armazena as ações em potencial esperando por execução e solução representa as hipóteses e decisões que o sistema gerou até um dado momento e as dependências relacionadas com outras decisões e hipóteses.

A base de conhecimento armazena regras, fatos e informações sobre o problema que poderão ser úteis na formulação da solução. Os fatos representam o conhecimento dos valores de algumas expressões, que são válidas para todas as consultas. As regras indicam como calcular o valor de certas expressões, e como proceder no caso dos valores de algumas expressões serem desconhecidos, etc.

O escalonador mantém controle da agenda e determina qual é a próxima ação pendente que deve ser executada.

O interpretador executa a ação escolhida da agenda aplicando a regra correspondente da base do conhecimento. O interpretador encadeia as regras entre si, para poder concluir os valores das expressões que são o objetivo do sistema combinando: o conhecimento contido na base de conhecimento com os dados fornecidos pelo usuário. Este encadeamento pode ser feito para-trás e para-frente, (Veja Figura 1).

O módulo de consistência é responsável pelo ajuste das conclusões obtidas, quando novos dados altera sua base de conhecimento.

O módulo explicador explica as ações do sistema para o usuário. Em geral responde a perguntas sobre porque certas conclusões foram obtidas ou porque algumas alternativas foram rejeitadas, permitindo ao usuário o exame do processo de raciocínio subjacente as respostas do sistema.

O Escalonador, Interpretador e o Módulo de Consistência são também denominados, muitas vezes, de máquina de inferência. Se quisermos fazer uma analogia com a atividade humana podemos dizer que a máquina de inferência corresponde à inteligência; a base de conhecimento, corresponde ao conhecimento que o especialista possui sobre o problema que vai resolver; e a memória de trabalho, é o lugar onde o especialista armazena as informações relacionadas com o problema concreto. Uma característica dos três componentes básicos é que, a máquina de inferência é um elemento permanente, podendo inclusive ser usado em outros sistemas baseados em conhecimento; a base de conhecimento é também um elemento permanente, porém específico de um determinado sistema especialista, e a memória de trabalho está vinculada a uma consulta concreta, e geralmente se conserva apenas durante o transcurso da mesma.

1.3.3 Compiladores Baseados em Conhecimento

Esta Seção aborda dois aspectos do problema: o primeiro vê o compilador como uma ferramenta na construção de um sistema baseado em conhecimento, e o segundo está relacionado com a elaboração do próprio compilador.

A idéia de um compilador inteligente é tornar possível o uso das técnicas de representação do conhecimento de uma maneira mais geral possível, enquanto se constrói e depura

um sistema baseado em conhecimento, e então aplicar um compilador inteligente com o objetivo de tornar a base de conhecimento usada eficiente a tal ponto, que compita com o código otimizado manualmente.

Dado um compilador deste tipo não há necessidade de sacrificar a flexibilidade por eficiência durante a etapa de construção e depuração de um sistema especialista. A base de conhecimento pode ser trocada a qualquer momento e recompilada quando necessário. O compilador pode também ser modificado para representar o conhecimento quando a arquitetura da máquina for substituída, ou quando os compromissos da representação se tornarem mais entendidos. Técnicas para fazer isto estão começando a ser exploradas, entretanto, não se encontra na literatura discussões sobre o assunto. Até agora, as discussões sobre eficiência têm sido no sentido de que os projetistas da base do conhecimento devem antecipar, como o conhecimento vai ser usado, e então, arranjá-lo para que seja representado eficientemente. Idéias de compiladores inteligentes, até o momento, são idéias mais especulativas que outras consideradas até agora, e muitas arestas devem ser resolvidas antes de torná-las práticas.

As pesquisas referentes à construção de compiladores baseados em conhecimento relatadas atualmente na literatura [WULF 80], [HARADVALA 84], [WARFIELD 88] concentram-se nos processos de geração e otimização de código, e na geração automática dos mesmos, talvez porque as técnicas de análise léxica e sintática já estejam bem solidificadas. Por exemplo, sabe-se em princípio ser possível usar uma técnica geral, porém, poderosa para efetuar a tradução. Não há uma necessidade real para a separação das análises, léxica e sintática, de fato a divisão entre elas em um compilador, muitas vezes, é efetuada arbitrariamente. 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 de análise léxica, todavia, a sua separação indroduz melhorias no sistema como um todo, tornando-o mais rápido.

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 como foi mostrado na Seção 1.3.1 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 sabem pouco em particular e sabem muito em geral. Assim, quando deparados com uma tarefa específica, métodos gerais devem "redescobrir" soluções que poderiam ser óbvias a um especialista. Nestas tarefas complexas, sabe-se que um conjunto de métodos complementares ou especialistas é mais eficaz que os métodos mais gerais. Ao contrário dos métodos gerais ou poderosos, um especialista (expert) simplesmente não consegue solucionar a maior parte dos problemas. Por outro lado, àqueles que ele soluciona, ele o faz bem e mais barato. Análise léxica baseada em máquina de estado-finito é um método especialista, 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 abrangente poderia.

Não é comum um projetista de compilador pensar em três ou quatro fases de um compilador como uma coleção de médodos especialistas quando se está escrevendo um compilador para uma linguagem e máquina alvo específicas. Tal conceituação é desnecessária. Entretanto, no projeto PQCC [WULF 80] ela foi essencial, pois foi através da análise cuidadosa de compiladores existentes, e na sua decomposição em fases, que se tornou possível encontrar trechos que poderiam ser parametrizados.

Wulf, 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".

Haradhvala et alii relata o sucesso obtido no uso de um sistema baseado no conhecimento, no processo de geração de código de compiladores. Para um compilador ser útil, ele deve gastar um tempo proporcional ao tamanho do programa que está sendo compilado. Para que isto acontecesse neste sistema foram introduzidas algumas otimizações além da redução do espaço de pesquisa. Muito embora as modificações no espaço de pesquisa tenham reduzido o poder do sistema, o código gerado pelo sistema baseado em conhecimento é melhor do que o produzido por muitos compiladores projetados usando técnicas convencionais para arquiteturas de máquinas e linguagens similiares.

1.4 PROBLEMA A SER RESOLVIDO E METODOLOGIA DE SOLUÇÃO

Nas Seções anteriores revimos, brevemente, metodologias de construção de compiladores e o papel do sistema de geração de compiladores na produção de um compilador. Foram mostrados a importância do uso de otimizadores no processo de compilação e o estado da arte com relação as mais recentes ferramentas criadas para o desenvolvimento dos mesmos, enfatizando os otimizadores locais. Na Seção 1.3 exibimos, uma visão geral da área de Inteligência Artificial e algumas tendências da pesquisa corrente na construção de otimizadores locais usando suas técnicas e metodologias. Foi mostrado, também, a importância de uma definição formal da arquitetura da máquina alvo. O objetivo foi prover o contexto para o problema, para o qual propomos buscar uma solução.

O problema proposto é o projeto de um Gerador de Otimizador Local (GOL) baseado em técnicas oriundas da Inteligência Artificial cuja solução incluirá:

1. O projeto de uma linguagem para descrever o conhecimento embutido nas arquiteturas de computadores.

2. O projeto de um Sistema Baseado em Conhecimento que a partir da descrição de

uma arquitetura de máquina específica gere, automaticamente, regras que possibilitem dirigir um otimizador local.

3. Projeto de um otimizador local fixo e dirigido por regras.

Propomos buscar a solução para o problema de acordo com a metodologia descrita a seguir.

A adequação da linguagem e do sistema propostos será verificada através do projeto e implementação de um Gerador de Otimizadores Locais (GOL). A entrada do GOL será a descrição da máquina e a saída o otimizador local dirigido por regras. Este otimizador local é fixo para qualquer arquitetura, sendo dirigido pelas regras geradas pelo GOL. A entrada para este otimizador será o código objeto e o resultado deverá ser sequências de instruções otimizadas da máquina alvo. As Figuras 4 e 5 apresentam o diagrama completo do GOL e e a utilização do produto obtido.

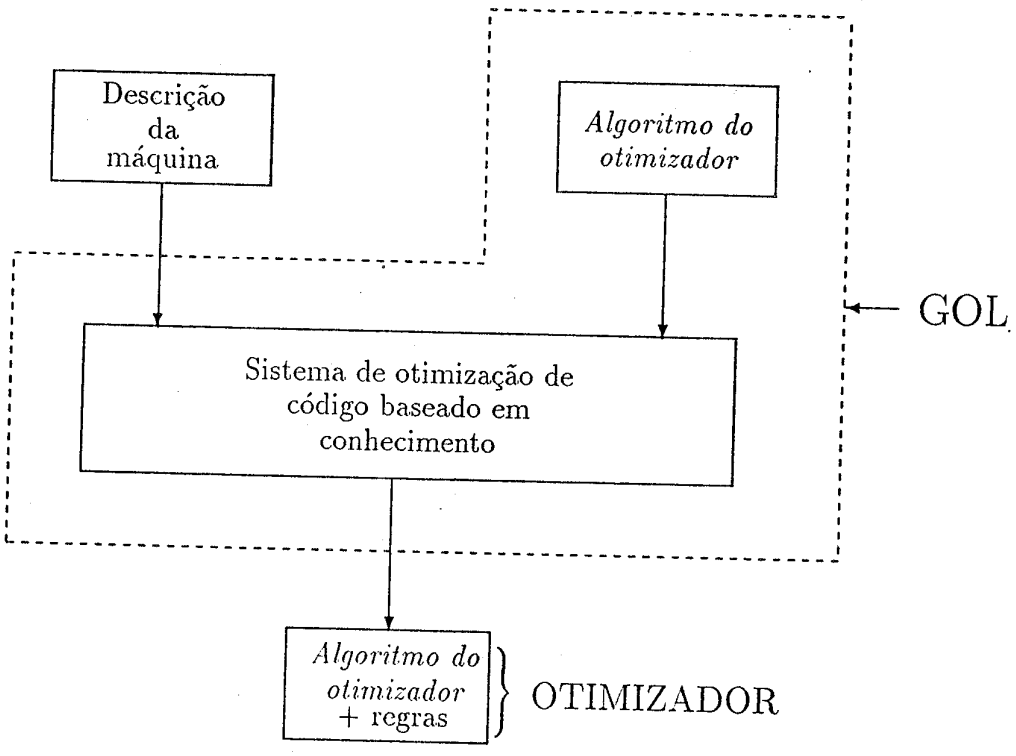

Figura 4: Sistema GOL.

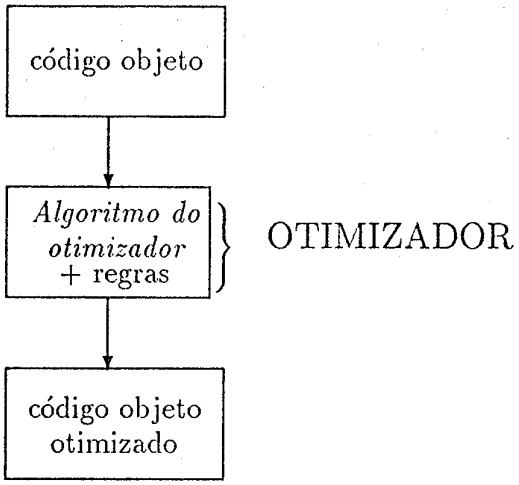

Figura 5: Uso do Otimizador

Faz parte deste projeto a avaliação da qualidade do código obtido, bem como as dificuldades advindas da metodologia adotada, ou da arquitetura da máquina alvo escolhida.

Existe uma série de questões que são levantadas referentes a linguagem proposta no Item 1, por exemplo: o que ela deve prover, quais são as suas características e quais os caminhos de solução possíveis com relação aos diferentes modos de endereçamento de uma dada arquitetura. Enfim, como sistematizar o conhecimento da máquina de uma forma clara e concisa.

Alguns trabalhos, [FRASER 89] e aqueles mostrados nas Seção 1.2.4, [DAVIDSON 80], [KESSLER 84], [WARFIELD 88], já foram efetuados no sentido de responder algumas destas questões.

Um consenso sobre qual linguagem é a melhor é difícil de se obter, porque todas elas basicamente diferem somente na notação e terminologia adotadas, tornando a seleção da "melhor" apenas uma questão de gosto. De qualquer forma, uma linguagem para ser usada em um sistema baseado em conhecimento visando um otimizador local, além de possuir construções que lhe permitam a definição das características fundamentais de uma arquitetura de máquina, tais como: os modos de endereçamento, instruções, efeitos colaterais etc., deve ser capaz de expressar regras, que estabeleçam os critérios necessários para que regras de otimização de código sejam produzidas. Para alcançar esta meta, o conhecimento embutido nas regras deve ser capaz de identificar o efeito de cada instrução, de combinar duas ou três sequências de instruções para formar seu efeito combinado, de reconhecer a ordem dos operandos em uma instrução, de decidir se duas instruções são adjacentes, (neste contexto consideramos duas instruções adjacentes se a primeira declara um valor usado pela segunda instrução). Para identificar o efeito de cada

instrução é necessário que para cada uma delas seja definido o seu formato de entrada, a equação semântica descrevendo sua função e o custo tempo e espaço. Como um caminho de solução, o que propomos para o Item 1 é um estudo das linguagens de descrição de arquiteturas existentes e a subsequente escolha por uma delas ou a proposta de uma nova.

Uma vez descrita a arquitetura da máquina, em uma dada linguagem, é importante estabelecer como esta informação será representada para ser usada pelo Sistema. Cogitamos de usar um sistema denominado quadro-negro (blackboard) [NII 86], mas esta hipótese, em princípio, está sendo descartada por ser considerada inadequada a este tipo de problema. Muito embora, o sistema de quadro-negro seja útil para problemas complexos e mal estruturados, ele normalmente é caro para ser construído e executado. Seria desperdício aplicar a abordagem de quadro-negro, quando métodos mais baratos são suficientes para solucionar o problema. A ocorrência da combinação de algumas das características enumeradas abaixo, em um problema, o tornaria um candidato apropriado para esta abordagem.

1. Um espaço de solução grande.
2. A presença de dados incertos e "ruidosos".
3. A variedade dos dados de entrada e a necessidade de integrar diversas informações.
4. A necessidade de conhecimentos independentes ou semi- independentes para cooperarem na elaboração de uma solução.
5. A necessidade de diferentes métodos de raciocínio, por exemplo, raciocínio para-frente e para-trás.
6. A necessidade de uma solução em evolução.
7. A necessidade de múltiplas linhas de raciocínio.

No problema em questão somente as características (3), (5) e (6) se fazem presentes, não justificando assim o uso de tal método.

Uma outra abordagem seria representar a descrição da arquitetura por meio de tabelas. Neste esquema poderia se ter um gerador de tabelas, que a medida em que a definição de máquina fosse sendo lida, o gerador iria construindo uma tabela com os recursos definidos e usados para cada instrução. Este processo de construção da tabela seria uma simples tradução da descrição da máquina para uma forma utilizável. O gerador de tabelas poderia por exemplo, a partir da definição da instrução produzir uma tabela indexada pelos operadores da máquina. Cada entrada na mesma incluiria os recursos usados e aqueles definidos, além das equações semânticas descrevendo a função de cada instrução. O trabalho, propriamente dito, do gerador de tabelas teria início quando ele começasse a pesquisar por instruções que pudessem ser combinadas em uma única e mais eficiente instrução. É exatamente neste ponto, que entra a elaboração do sistema baseado em

conhecimento do GOL, um dos pontos centrais na solução do problema proposto, com o objetivo de minimizar a quantidade de trabalho necessária na elaboração desta tarefa.

Nos otimizadores de código descritos na Seção 1.2.4, em particular, PO, PEEP e HOP, as técnicas usadas para encontrar as instruções sujeitas à otimização são fortemente baseadas em casamento de padrões (pattern-matching) e pesquisas, ou necessitam do auxílio de um conjunto de programas de treinamento. O Sistema Baseado em Conhecimento efetua a pesquisa e o casamento de padrão automaticamente e não precisa de um programa de treinamento.

Na solução proposta, o GOL deve a partir da definição de uma arquitetura armazenada em uma forma qualquer, incluindo um conjunto de regras de otimização e do algoritmo do otimizador, produzir como resultado um outro conjunto de regras que possibilite dirigir um otimizador local tradicional fixo. Este novo conjunto seria automaticamente inferido através do processo de combinação e subsequente troca de instruções. As regras contidas na base de conhecimento indicariam metas que devem ser cumpridas para que tal processo seja efetuado. À medida que a substituição de instruções fosse sendo efetuada, seria formado um esboço textual da troca de tal forma que as regras resultantes seriam formadas automaticamente através da parametrização destes trechos. A parametrização substituiria cada constante distinta em uma instrução com um padrão de variável diferente, por exemplo, poderia ser usado uma notação do tipo "%i", onde i representaria operandos sensíveis ao contexto.

As instruções, no código objeto a ser otimizado, poderiam ser representadas na forma de triplas. Esta representação é análoga ao código de linguagem de montagem convencional: o primeiro campo define o que a instrução faz e os outros campos descrevem os operandos das instruções.

O otimizador poderia usar "hashing" para interpretar as regras. Para evitar manipulação de "strings", ele poderia separar cada padrão ou esqueleto de instrução de seus operandos a medida que os fosse lendo, e armazená-los em uma tabela "hash". O mesmo procedimento seria adotado para as regras. Com isto os dois "strings" se tornam idênticos e poderiam ser comparados pelo teste de endereços na tabela "hash". Primeiramente, comparar-se-ia os dois esqueletos da entrada com as duas primeiras linhas da regra, comparando dois pares de apontadores. Depois o otimizador verificaria a consistência dos operandos entre as duas instruções comparando o primeiro operando da primeira instrução com o primeiro operando da segunda instrução, novamente, pela comparação de endereços. Finalmente, se toda comparação fosse bem sucedida, uma substituição de instrução seria efetuada. Seu esqueleto seria a última linha da regra. Substituiria-se os operandos de acordo com aqueles das instruções originais, e esta instrução representaria a substituição para as duas instruções originais. Caso acontecesse alguma falha na comparação, nova regra seria ativada.

Em nosso trabalho, as abordagens delineadas acima serão investigadas como subsídio para se obter uma solução para o problema proposto.

1.5 CONCLUSÃO

O uso de um sistema baseado em conhecimento na geração de compiladores otimizantes de código introduz uma série de vantagens:

1. A quantidade de trabalho necessário para um programa para construir e manter um otimizador local redirecionado é reduzido porque o mesmo não necessita de um conjunto de programas de treinamento o qual deveria ser reescrito para cada máquina nova.
2. Para redirecionar o Sistema para outra arquitetura basta trocar a descrição das instruções e dos modos de endereçamento.
3. Os Sistemas Baseados em Conhecimento podem fazer inferências a partir de fatos conhecidos. Isto significa que menos código deve ser escrito para um otimizador local redirecionável.
4. Regras adicionais podem ser incluídas ao Sistema sem alterar o resto do programa.

As principais contribuições deste trabalho compreendem o projeto de uma linguagem para descrever o conhecimento embutido nas arquiteturas de computadores e um mecanismo para geração automática de regras que possibilitem a realização de otimizações locais.

Os principais resultados que se pretende obter após a solução do problema proposto são:

1. Demonstrar a viabilidade ou não do método adotado.
2. Desenvolvimento de novas técnicas para a produção de otimizadores de código redirecionáveis.
3. Produção de um sistema que reduza o custo de implementação de otimizadores de código redirecionáveis.

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.

[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.

[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.

[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.

[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 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.

[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., Watèrman 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.

[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., 2SIS - 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.

[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.

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

[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 Medical Consultations: MYCIN, American Elsevier, New York, 1976.

[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.

[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.