# visual Klar - Um Ambiente Integrado para Depuração Simbólica de Código Intermediário Machina

**Projeto de Iniciação Científica**

**Área: Linguagens de Programação**

**Orientadora: Profa. Mariza Andrade da Silva Bigonha**

Equipe: Profa. Mariza Andrade da Silva Bigonha
Prof. Roberto da Silva Bigonha
Fabíola Fonseca (doutoranda)
Kristian Magnani dos Santos (mestrando)

DCC – ICEx - UFMG

Belo Horizonte, 26 de novembro de 2004

## 1. Motivação

Métodos informais de definição de linguagens de programação carregam uma imprecisão semântica que pode resultar em uma descrição inconsistente ou ambígua, sendo, por isso, inadequados para descrever linguagens de programação. Métodos formais são fortes onde os métodos informais apresentam problemas, garantindo precisão e diminuindo as possibilidades de inconsistências em definição de linguagens. Porém, eles também apresentam problemas: podem tornar a leitura e escrita tarefas complicadas, e, além disto, são restritos pela falta de ferramentas que sejam fáceis de utilizar e, ao mesmo tempo, gerem código eficiente.

Máquinas de estado abstratas (ASM, Abstract State Machines), introduzidas por Yuri Gurevich [5] como um novo modelo computacional, ao contrário de muitos métodos formais de especificação, utiliza conceitos simples e bem conhecidos, tornando a leitura e a escrita da especificação bastante direta. Originalmente proposta com o objetivo de prover semântica operacional para algoritmos de uma forma mais natural que a máquina de Turing, este modelo tem sido usado com sucesso para formalizar sistemas de tempo real, seqüenciais, paralelos e distribuídos, além de arquitetura de computadores [2,3,4,5,6].

A idéia do modelo é realizar simulação de algoritmos via transições de estado. Para descrever um algoritmo, inicialmente define-se um estado inicial e uma regra de transição. Estado é um conjunto dos nomes de funções e relações, juntamente com as suas interpretações. O conjunto dos nomes de funções e relações é o vocabulário do estado. A interpretação de um nome de função ou relação é um mapeamento dos nomes do vocabulário na respectiva função ou relação. As principais regras são: atualização; regras guardadas; regras bloco.

Uma computação em ASM causa uma mudança de estado, sendo descrita por uma regra de transição que modifica a interpretação de nomes de função do vocabulário do estado. Esse mecanismo de transição de estados por regras de transição assemelha-se a comandos de programação imperativa. Como resultado, uma especificação ASM aparenta um programa imperativo, sendo fácil de entender e de executar. A principal diferença entre as duas formas de programação é a ausência de iteração em ASM, que é obtida usando um conceito implícito de execução repetitiva da regra de transição.

ASM define uma notação de especificação executável e provê uma base formal para uma notação, que pode ser usada tanto como uma linguagem de especificação quanto de programação de alto nível. Porém, para se tornar uma linguagem de programação real, a notação precisa, além de outros fatores, apresentar um código executável eficiente, que leve em consideração aspectos como tempo e espaço de memória. O atual estado da arte sobre ferramentas de suporte para este modelo oferece implementações com pouca ou nenhuma capacidade de otimização.

Com o objetivo de fazer uso das vantagens oferecidas por este modelo, projetou-se, no Departamento de Computação da UFMG, Machĭna [10], uma linguagem de especificações ASM. Machĭna tem suporte à modularidade e permite construções de alto nível. É uma linguagem fortemente tipada, com a facilidade de inferência de tipos em tempo de compilação. Apresenta possibilidade de definir invariantes para execução da regra de transição da máquina abstrata, bem como uma noção de multiagentes introduzidos de maneira simples e direta.

Um programa em Machĭna consiste em um ou mais módulos executados por agentes ou processos concorrentes. Cada agente executa a regra de transição definida em algum módulo. Na regra de transição pode haver a criação de novos agentes, sendo que os agentes iniciais são disparados em um módulo inicial denominado módulo Machĭna. Uma regra de transição terá sua execução repetida enquanto ocorrerem mudanças de estado da máquina.

O processo de compilação de programas escritos em linguagens baseadas em ASM, em geral, consiste na tradução do programa fonte para uma linguagem imperativa, a qual posteriormente deve ser traduzida para um código de baixo nível de uma máquina real. Devido às diferenças entre o paradigma de linguagens baseadas em ASM e as máquinas de Von Neumann, programas desenvolvidos em linguagens baseadas em ASM apresentam dificuldades particulares na geração de um código eficiente.

Gerar um código ótimo é um problema matemático indecidível [1], mas que encontra soluções interessantes em técnicas heurísticas. A escolha de uma boa heurística é difícil, mas muito importante para a eficiência do código gerado e, portanto, deve levar em consideração o maior número de informações possível, como por exemplo a máquina para qual se está gerando código, as características principais da linguagem fonte.

A Figura 1, extraída de [14] mostra o processo de compilação de um programa em Machĭna, onde $L_{MIR}$ representa a linguagem intermediária. $L_{MIR}$ [9, 14] é uma linguagem simples, estruturada como uma árvore, cujas instruções possuem características próximas às instruções da linguagem

fonte desta compilação, Machĭna. Ou seja, instruções MIR tem características de programação ASM.

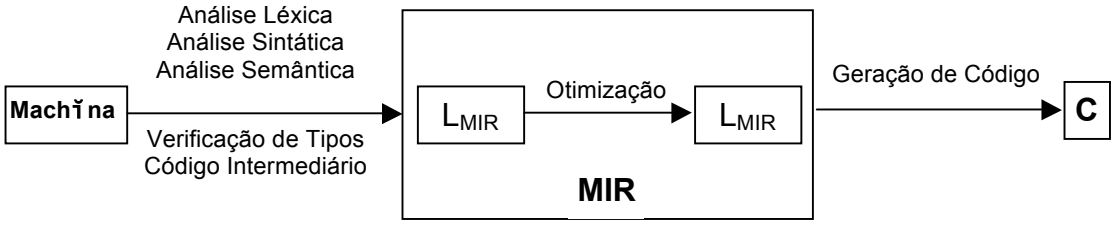


**Figura 1: Compilador de Machĭna**

Um programa em linguagem $L_{MIR}$ denota uma regra de transição, que pode ser representada diretamente por estruturas de dados na memória. Existem duas estruturas de dados principais: uma árvore de nomes, com informações oriundas da tabela de símbolos gerada pelo compilador durante as fases de análise sintática e semântica, e uma árvore cujos nodos são as instruções MIR.

> Outros componentes da arquitetura abstrata de MIR são: memória de dados, memória de código, vetor de estado, registradores auxiliares, pilha de contadores de programa, ambiente, etc.

A arquitetura MIR é uma infraestrutura projetada para servir como base para compiladores concorrentes ASM. O projeto MIR, Machĭna Intermediate Representation, originou-se na necessidade de uma linguagem intermediária durante a compilação de código Machĭna para código C. O projeto foi ampliado e atualmente consiste em um projeto separado da compilação de Machĭna. Com o projeto **Arquitetura MIR** será possível realizar experiências com linguagens baseadas no modelo ASM.

O desenvolvimento de uma infraestrutura geral para compiladores ASM tem por base a possibilidade de realizar experiências com linguagens orientadas ao modelo ASM. A arquitetura MIR foi projetada para suprir tal necessidade. Além disso, possui a capacidade de execução concorrente, útil na implementação de algoritmos concorrentes.

O objetivo principal da Infraestrutura MIR é produzir código ANSI C a partir de uma especificação MIR, permitindo, então, um desenvolvimento rápido para compiladores ASM. A Figura 1 mostra o contexto de MIR.

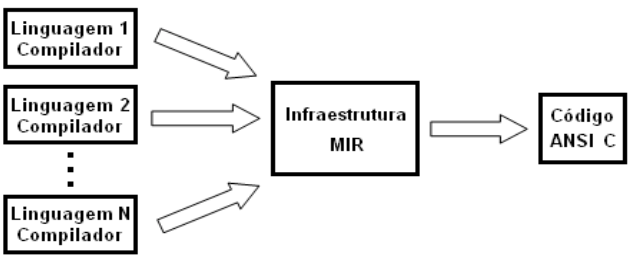

**Figura 1: Contexto de MIR**

Outro ponto importante sobre MIR é o fato de ser projetado para ser otimizado. As otimizações propostas no projeto MIR não sobrepõem as otimizações comumente realizadas por um compilador C, pois elas são otimizações que levam em consideração as características do modelo ASM.

A Infraestrutura MIR são todas as classes e componentes de software que compõe a infraestrutura implementada. A expressão Arquitetura MIR se refere à estrutura geral de uma especificação ASM usando a Infraestrutura MIR, podendo ser vista como a "linguagem" da infraestrutura. E, se a Arquitetura MIR é a linguagem, uma Especificação MIR é um "programa" escrito nessa linguagem.

## 1.1. Agentes, Modelos e outros elementos

Uma especificação MIR, como mostrada na Figura 2, é composta por agentes, cada um de um tipo, e por um Espaço de Nomes Globais comum a cada um dos agentes que pertencem à especificação MIR. O Espaço de Nomes Globais é uma tabela para as ações e funções estáticas, derivadas, dinâmicas e externas. Nesta tabela é possível identificar se a informação é uma ação ou função, assim como o tipo da função. Funções estáticas são aquelas cujos valores não mudam ao executar as regras de atualização. Tais funções são definidas por expressões parametrizadas e permanecem inalteradas durante toda a execução da arquitetura MIR. Não é permitido chamar funções dinâmicas de dentro das definições das funções estáticas.

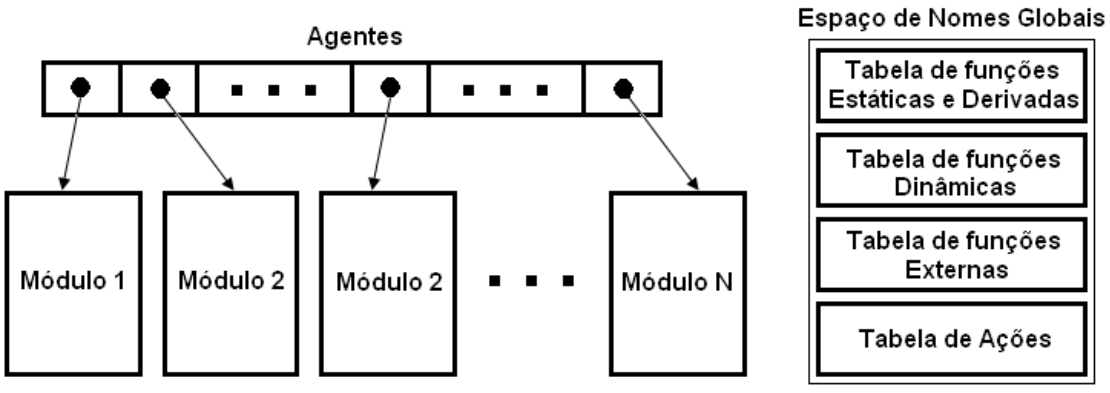


**Figura 2: Arquitetura MIR**

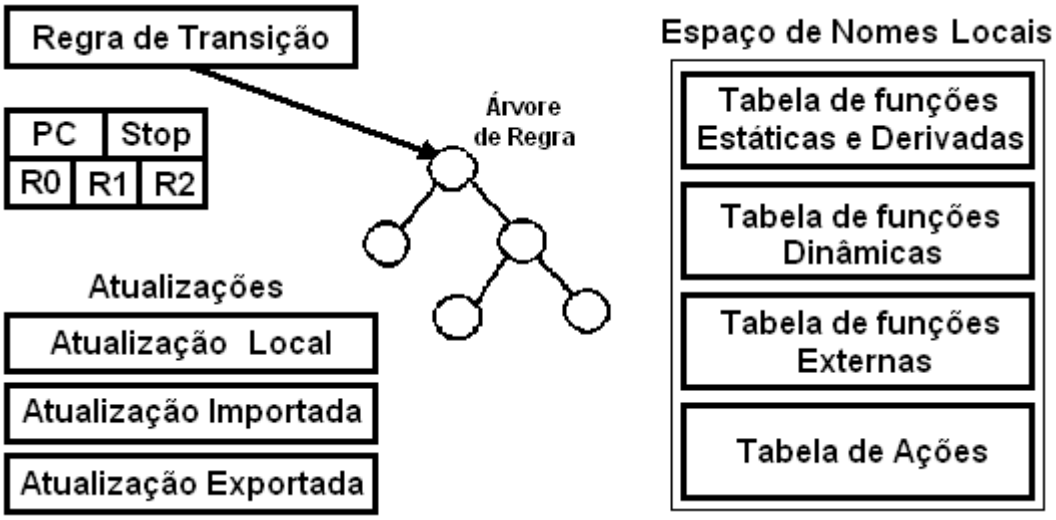


**Figura 3: Um módulo de MIR**

Especificamente, nossa proposta consiste na especificação e desenvolvimento de um ambiente visual para a depuração simbólica de programas escritos em MIR, denominado *Visual Klar*.

## 2. Objetivo

O objetivo do *Visual Klar* é oferecer ao programador MIR facilidades de usabilidade, tais como:

- *pretty print* de arquivos de serialização MIR,
- identificação de erros de sintaxe,
- *auto-complete* com contexto dinâmico,
- representação visual da MIR como uma árvore,
- execução passo a passo da estrutura MIR,
- acompanhamento de valores de funções dinâmicas por meio de *watchs*,
- outras funções típicas de uma IDE de programação para a depuração simbólica de programas escritos em MIR.

Este ambiente deve ser escrito em Java, e deve fazer uso do arcabouço Klar [9], descrito na Seção 3.

## 3. Contexto

O projeto de iniciação científica proposto está inserido no contexto do projeto "*Klar - Um Arcabouço para Otimizações em Máquinas de Estado Abstratas*" [9]. O principal objetivo do projeto do Klar é fornecer um arcabouço para otimização

de representações intermediárias,
especificamente MIR, onde as otimizações são facilmente plugáveis. Além disso, este trabalho define: (a) o padrão para arquivos de MIR serializada, (b) implementa a estrutura MIR, (c) define alguns padrões de projeto *visitors* adicionais [15,9], que permitem serializar a estrutura MIR, executá-la diretamente, compilá-la para código ANSI C e obter uma representação visual desta estrutura.

A integração do *Visual Klar*, proposto neste texto, com o *Klar* é muito simples. O *Visual Klar* utilizará o arcabouço Klar como sua base. Especificamente, não será necessário o desenvolvimento de rotinas de compilação, otimização de código ou serialização de estruturas MIR, aproveitando-se o arcabouço do *Klar* como um todo. O ponto central do *Visual Klar* é o desenvolvimento de facilidades visuais para a execução e depuração simbólica de uma estrutura MIR, como enunciadas na Seção 2.. Estas facilidades visuais tem impacto positivo sobre a usabilidade do *Klar*.

## Referências


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[2] E. Borger and J. Huggings. *Abstract State Machine 1988-1998: Commented ASM Bibliography.* Bulletin of EATCS, 64: 105-127. Fevereiro, 1998.

[3] E. Borger and D. Rosenzweig. *A Mathematical Definition of Full Prolog.* In Science of Computer Programming, volume 24, pp 249-286, 1994.

[4] Y. Gurevich. *Evolving algebras: An Attempt to discover semantics.* Bulletin of the European Association or Theoretical Computer Science, 43:264-284, 1991

[5] Y. Gurevich. *Evolving algebras 1993: Lipari guide.* In E. Borger, editor, Specification and Validation Methos, pages 9-36. Oxford University Press, 1995

[6] Y. Gurevich and J. Huggings. *The Semantics of the C Programming Language.* In E. Borger, H. Kleine , G. Jager, S. Martini, and M. M. Richter, editors, Computer Science Logic, volume 702 of LNCS, pp 274-309, Springer, 1993.

[7] P. W. Kutter and A. Pierantonio, *The Formal Specification of Oberon.* Journal of universal computer science, Vol. 3, No 5 (1997), pp 443-503, Springer.

[8] C. Wallace, *The Semantics of the Java Programming Language: Preliminary Version.* Univ. of Michigan EECS Department Technical Report CSE-TR-355-97.

[9] Santos, K. M, *Proposta de Dissertação de Mestrado*. Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. 2004.

[10] Tirelo, Fábio, *Uma Ferramenta para Execução de um Sistema Dinâmico Discreto Baseado em Álgebra Evolutivas*, Dissertação de Mestrado, DCC-ICEx-UFMG, 2000.

[11] The Eclipse Project Homepage: www.eclipse.org.

[12] Vartan Piroumian, *Java Gui Development*. Sams Publisher. 1st edition. 1999.

[13] James Elliott, Robert Eckstein, Marc Loy, David Wood, Brian Cole, Java Swing. O'Reilly. Second Edition. 2002.

[14] Fonseca, Fabíola, *Linguagem Intermediária MIR*, Relatório Técnico 001/2004 do Laboratório de Linguagens de Programação, DCC-ICEx, UFMG a ser publicado, 2004 .

[15] Metsker, Steven John, *Padrões de Projeto em Java*, Editora Bookman, Tradução: Werner Loeffler, Leitura Final: André Luís de Godoy Vieira, 2002.

**visual Klar - Um Ambiente Integrado para Depuração Simbólica de Código Intermediário Machina**

**Projeto de Iniciação Científica**

**Área: Linguagens de Programação**

**Orientadora: Profa. Mariza Andrade da Silva Bigonha**

## 1. Plano de Orientação de Iniciação Científica

Para atingir nosso objetivo, produzir um ambiente para a depuração simbólica de programas escritos em MIR, este projeto foi dividido em três fases. A primeira fase consiste no estudo dos conceitos relacionados à ASM e Machina e sobretudo MIR,a linguagem intermediária de Machina. Serão estudadas também nesta fase ferramentas de visualização. A segunda fase consiste na especificação e implementação do ambiente de depuração proposto. Fazem parte do projeto também, a elaboração de uma massa de testes e produção de manuais do usuário e do sistema que correspondem à terceira fase. O manual do sistema será escrito ao longo do desenvolvimento do projeto. Para facilitar a visualização do trabaho, a Seção 6 apresenta o cronograma de desenvolvimento do mesmo.

## 2. Descrição das Fases

Fase 1: Revisão da Literatura

- Estudo dos conceitos e princípios do paradigma ASM ([2] – [8])
- Estudo de Machina ([10])
- Estudo de MIR ([9],[14])
- Estudo de ferramentas de visualização ([11], [12], [13])
- Elaboração de um relatório parcial.

Fase 2: Desenvolvimento do ambiente para depuração simbólica de programas MIR

- Especificação do ambiente
- Desenvolvimento e implementação
- Elaboração de um relatório parcial

Fase 3: Testes e elaboração de manuais

- Testes
- Elaboração do manual do usuário
- Elaboração do manual final do sistema – feito durante todo o desenvolvimento do projeto a partir dos relatórios parciais.

## 3. Cronograma

Fase 1: 15 de março de 2006 a 30 abril de 2006
Fase 2: 02 maio de 2006 a 30 de setembro de 2006
Fase 3: 01 de outubro de 2006 a 20 de fevereiro 2007

# visual Klar - Um Ambiente Integrado para Depuração Simbólica de Código Intermediário Machina

**Projeto de Iniciação Científica**

**Área: Linguagens de Programação**

**Orientadora: Profa. Mariza Andrade da Silva Bigonha**

## 1. Metodologia de Acompanhamento e de Avaliação

Na primeira fase do projeto haverá duas reuniões semanais, de aproximadamente uma hora, com a professora orientadora, a equipe do projeto para acompanhamento dos estudos do bolsista, discussão de funcionamento e relato das dificuldades encontradas. Na segunda fase, o bolsista desenvolverá de forma independente seu trabalho, reunindo-se uma vez por semana, durante uma hora, com a professora e sua equipe, para avaliação das etapas elaboradas e discussão daquelas a serem desenvolvidas. Esta metodologia será empregada durante todo o desenvolvimento do trabalho.