Compilador para linguagem Monga

Trabalho Final de Compiladores

Guilherme Dantas

PUC-Rio

Dezembro de 2020

Sumário

Monga

Monga é uma linguagem de programação baseada em C.
Para a especificação completa da linguagem, visite
`http://www.inf.puc-rio.br/~roberto/comp/lang.html`.

Pipeline

É fácil perceber que o processo de compilação de um programa Monga é composto de uma sequência de processos, como veremos a seguir.

Análise léxica

Lex é um programa presente na maioria dos sistemas UNIX, que permite que bibliotecas C consumam (**scanning**) uma entrada de caracteres e reconheçam expressões regulares como *tokens*. Esta entrada provém do programa Monga.

Análise sintática

Yacc é outro programa comum em sistemas UNIX, que permite que bibliotecas C realizem o **parsing** de uma entrada de *tokens* (provinda do Lex) segundo uma gramática livre de contexto. Comumente é usado para criar árvores sintática abstratas.

Análise semântica

Esta etapa envolve analisar a árvore sintática abstrata construída no passo anterior de forma a alcançar principalmente os seguintes objetivos:

- ligar referências às definições referenciadas
- realizar a propagação e chegagem de tipos

Geração de código LLVM

Baseando-se em exemplos de programas C compilados com Clang[1], mapeou-se elementos da linguagem Monga com código LLVM. Foi ainda necessário cumprir os seguintes requisitos:

- identificar variáveis e expressões (variáveis em SSA)
- gerar código LLVM para definições implícitas de funções e de literais

Example

```
[compilador-monga] < programa.mon > programa.ll
```

[1] `clang -emit-llvm -S`

Geração de código *assembly*

O programa `llc` compila o código LLVM (`.ll`) para código *assembly* (`.s`). Pode-se ainda, antes desse passo, otimizar o código LLVM (`opt`).

Example

```
llc programa.ll
```

Geração de código de máquina

Escolhe-se um compilador para compilar o programa de *assembly* para linguagem de máquina (formato executável).

Example

```
[compilador-assembly] -no-pie programa.s -o programa
```

Observação

A flag `-no-pie` instrui compiladores como `gcc` e `clang` a não produzirem um executável com código independente de posição.

Highlights

Serão destacados alguns pontos positivos referentes à implementação do compilador, que podem servir de inspiração para projetos futuros.

- organização do código
- *framework* de testes unitários
- documentação interna
- modularização dos componentes
- detecção de vazamentos de memória

Organização

O código C foi dividido entre os seguintes módulos.

- **utils** - Funções e macros úteis (biblioteca)
- **l** - *Scanner* (Lex) (biblioteca)
- **ldb** - Depurador do *scanner* (Lex) (executável)
- **ast** - Árvore Sintática Abstrata (biblioteca)
- **y** - *Parser* (Yacc) (biblioteca)
- **ydb** - Depurador do *parser* (Yacc) (executável)
- **llvmdb** - Depurador do compilador para LLVM (executável)

Framework de testes unitários

Todo *debugger* possui casos de testes que consistem em:

- entrada (programa Monga)
- saída esperada

A saída difere de natureza, dependendo do tipo de *debugger*.

- ldb - *tokens* reconhecidos
- ydb - *abstract syntax tree* em formato parecido com Lisp
- llvmdb - código LLVM

Ao rodar os testes unitários, é comparada a saída esperada com a obtida[2], esperando-se que sejam idênticas.
Essa *framework* foi implementada por meio de *scripts* bash.

[2] `diff [esperada] [obtida]`

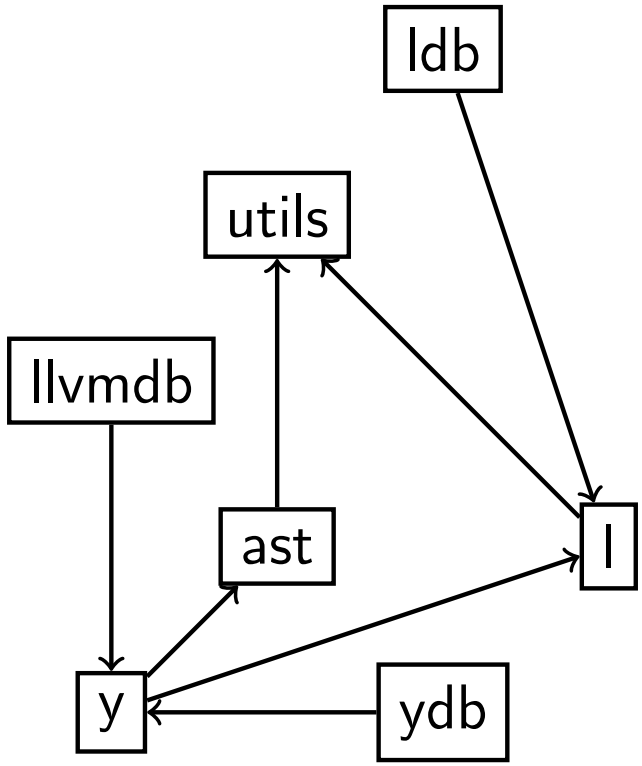
Documentação interna

Como forma de formalizar conceitos-chave da implementação, foram documentados em formato Markdown os seguintes tópicos.

- Árvore Sintática Abstrata
- Propagação e checagem de tipos
- *Binding* de referências a definições
- Geração de código LLVM

Modularização dos componentes

Como boa prática de programação, a estrutura do código segue uma lógica modularizada.



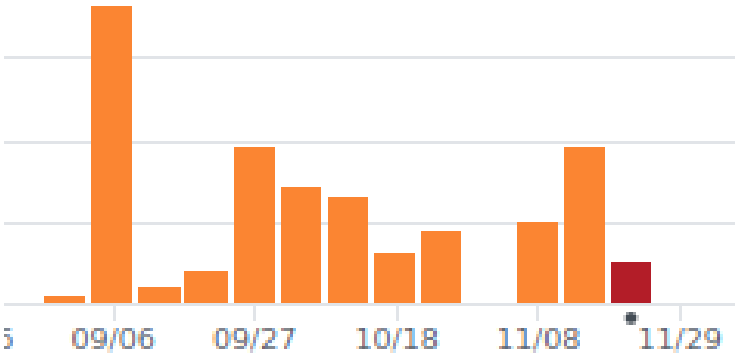
Detecção de vazamentos de memória

Esta funcionalidade está presente no módulo **utils** e é usado por todos os outros módulos, direta ou indiretamente. Funciona da seguinte forma:

- Toda alocação dinâmica de memória adiciona um nó com o endereço do espaço à lista de espaços alocados. Toda desalocação procura remover o nó que aponta para o mesmo endereço de memória.
- Ao final do programa, certifica-se que a lista está vazia.
- São armazenados ainda outros metadados como nome do arquivo e número da linha onde ocorreu a alocação dinâmica de memória, de forma que fontes de vazamentos sejam facilmente detectados.

Métricas

- 4834 linhas de código C, Lex e Yacc
  - `find src -name '*.[chly]' | xargs wc -l`
- 153 casos de testes unitários
  - ldb - 42
  - ydb - 77
  - llvmdb - 34
- 138 commits



  - `https://github.com/guidanoli/monga`