Otimizações em LLVM

Projeto 3:

- Implementar a otimização Dead Code Elimination no LLVM
  - Seu passo receberá o nome de **`-dce-p3`**

- **Material:** duas versões

  - Versão 1: Caso **não possua** 'llvm-config'
    - **llvm-p3**
      - Código fonte e objetos do LLVM para o Projeto 3
    - **llvm-pass**
      - Um passo que imprime o nome das funções (-hello)
      - Um passo que emite o CFG do programa (-printCFG)

  - Versão 2: Caso **possua** 'llvm-config'
    - **llvm-pass-v2**
      - Um passo que imprime o nome das funções (-hello)
      - Um passo que emite o CFG do programa (-printCFG)

Versão 1: Pacote llvm-p3

- Caso não tenha ‘llvm-config’, precisaremos:
  - do código fonte do LLVM
  - e do código objeto do LLVM (fonte já compilado)

- As máquinas do lab possuem apenas os executáveis e a libLLVM-3.3.so. Os objetos não estão disponíveis.

- Baixem este pacote do LLVM feito para o P3:
  - http://www.ic.unicamp.br/~maxiwell/cursos/mc911/projeto3/llvm-p3.tar.bz2
  - Contém o código fonte e os objetos compilados do LLVM-3.3

- No executável “drive”, coloque o caminho deste pacote em:
  - PATH_PACKAGE_P3=

Versão 1: Estrutura do pacote llvm-pass

- build:
  - Diretório que conduz a compilação e ligação com o LLVM
- p3:
  - Diretório dos fontes das suas otimizações
  - Cada otimização deve ter seu próprio arquivo .cpp
- Release:
  - Diretório que contém o objeto **P3.so** gerado após a compilação
  - **P3.so** contém todas as suas otimizações
- tests:
  - Diretório com exemplos de código em .ll para otimizar
- drive:
  - Bash script que conduzirá a compilação

Versão 2: Estrutura do pacote llvm-pass-v2

- p3:
  - Diretório dos fontes das suas otimizações
  - Cada otimização deve ter seu próprio arquivo .cpp
- Release:
  - Diretório que contém o objeto **P3.so** gerado após a compilação
  - **P3.so** contém todas as suas otimizações
- tests:
  - Diretório com exemplos de código em .ll para otimizar
- drive:
  - Script simples que chama o Makefile
- Makefile:
  - Compila suas otimizações fazendo uso do ‘llvm-config’

Comandos úteis

```
# Para instalar seu projeto, corrija a var PATH_PACKAGE_P3
./drive install # comando existente apenas na 'Versão 1'

 # Para compilar suas otimizações presentes em "p3"
 ./drive compile

 # Para limpar o projeto
 ./drive clean

 # Compilando arquivos .c para LLVM-IR
 clang -emit-llvm -S input.c -o output.ll
```

Comandos úteis

```
# Para utilizar a otimização que está em
# "p3/Hello.cpp" no código LLVM-IR "tests/BubbleSort.ll"
./drive opt -hello tests/BubbleSort.ll > BubbleSort.opt


# ou diretamente via comando "opt" do LLVM
opt -load Release/P3.so -hello tests/BubbleSort.ll > BubbleSort.opt


# O resultado "BubbleSort.opt" está em bitcode, que é um
# LLVM-IR compilado. Transformando bitcode em LLVM-IR ASCII:
llvm-dis BubbleSort.opt
```

Primeiros passos Versão 1

- Baixar e descompactar o ‘llvm-p3’ e ‘llvm-pass’
- Em ‘llvm-pass/drive’, colocar o caminho de ‘llvm-p3’ em PATH_PACKAGE_P3
- Instalar o pacote
  - **./drive install**
- Compilar os passos
  - **./drive compile**
- Aplicar o passo no programa BubbleSort
  - **opt -load Release/P3.so -hello tests/BubbleSort.ll > BubbleSort.opt**
- Executar o programa otimizado
  - **lli BubbleSort.opt**

Primeiros passos Versão 2

- Baixar e descompactar o ‘llvm-pass-v2’
- Compilar os passos
  - **./drive compile**
- Aplicar o passo no programa BubbleSort
  - **opt -load Release/P3.so -hello tests/BubbleSort.ll > BubbleSort.opt**
- Executar o programa otimizado
  - **lli BubbleSort.opt**

Dicas de Implementação

- Ler a documentação da API C++ do LLVM - Parte 1
  - Core Classes
  - Value class
  - User class
  - Function class
  - Instruction class
  - BasicBlock class
  - Iterando Instructions e BBs dentro de Functions

- **BasicBlock** (labels) é subclasse de **Value**
- **Instructions** e **Functions** são subclasses de **User**
- **User** é subclasse de **Value** e contém uma lista de operandos

- **SSA**: Única definição
  - Um operando de uma instrução é:
    - um apontador para a sua instrução de definição; ou
    - um argumento

```
%tmp0 = load i32* %a
%tmp1 = load i32* %b
%tmp2 = add i32 %tmp0, %tmp1
```

Instrução %tmp2

```
Instruction *I = /* instrução de um Basic Block */
errs() << "INS:" << *i << '\n';
for (Instruction::op_iterator o = i->op_begin(), oe = i->op_end(); o != oe; ++o) {
    Value *v = *o;
    if (isa<Instruction>(*v) || isa<Argument>(*v)){
        errs() << "\tOPE:" << *v << '\n';
        }
}
```

Testa se o operando é ponteiro para uma instrução ou para um argumento

Output:

```
INS:  add i32 %tmp0, %tmp1
      OPE:  %tmp0 = load i32* %a
      OPE:  %tmp1 = load i32* %b
```

Dicas de Implementação

- Ler a documentação da API C++ do LLVM - Parte 2
  - isa, cast e dyn_cast
  - mayHaveSideEffects
  - eraseFromParent
  - def-use e use-def chains

- Leiam o manual sobre implementação de passos

- Use C++ STL para melhorar sua produtividade em C++
  - Contém **sets, maps, lists** e **vectors**

Dead Code Elimination

- Uma instrução é trivialmente viva:
  - se seu comportamento gerar efeitos colaterais (*mayHaveSideEffects*)
  - se for um terminador (*TerminatorInst*)
  - se for instrução de debugging (*DbgInfoIntrinsic*)
  - se for instrução de exceção (*LandingPadInst*)
  - se for usada por outra instrução que está viva

- Códigos úteis:
  - Passo: Liveness sobre a IR do LLVM (não funciona no LLVM-3.3)
  - Passo: CFG Analysis sobre a IR do LLVM (não funciona no LLVM-3.3)

Analise Externa ao Passo

- Divisão de tarefa entre vários passos/análises
- Deixa o código mais organizado
- Mantém seu código dentro do padrão LLVM

AnalysisUsage e getAnalysis

```
#include "GenKill.h"
using namespace llvm;
namespace {
  struct Hello : public FunctionPass {

    Hello() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
        GenKill &GK = getAnalysis<GenKill>();
        ...
    }

    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
        AU.addRequired<GenKill>();
    }
  }
}
```

GK pode chamar métodos de GenKill

Chama o método Run de GenKill

Como fica GenKill.h?

```
namespace  {
        DenseMap<const Instruction*, int> instMap;
        struct Aux { int i, j; };
}

namespace llvm {
        struct GenKill : public FunctionPass {
        private:
                void compute1(...);                          // implementação no KillGen.cpp
        public:
                static char ID;
                GenKill() : FunctionPass(ID) {}
                virtual bool runOnFunction(Function  &F);    // implementação no KillGen.cpp
        };
}

// As duas linhas abaixo devem ser colocadas no KillGen.cpp
// char llvm::GenKill::ID = 0;
// RegisterPass<GenKill> X("genkill", "Live vars analysis", false, false);
```

Estruturas auxiliares devem ter escopos anônimos

Estruturas usadas em addRequired e getAnalysis devem pertencer ao escopo do LLVM

A atribuição do ID e o registro da estrutura deve ficar no arquivo .cpp