Compilertransformaties op LLVM IR niveau

Contactinformatie:
Adres *ELIS (verdieping -3), Technicum (blok I), Sint-Pietersnieuwstraat 41*
E-mail *compiler@elis.ugent.be*

BELANGRIJK: Het indienen van de oplossingen dient steeds te gebeuren via het Dropbox-menu op Minerva, dit zowel voor een tijdelijke versie op het einde van elk practicum als de finale versie voor onderstaande dealine. Gebruik het `make_tarball.sh` script om een archief te genereren, en zorg dat het naar de begeleiders verzonden wordt.

DEADLINE: 7 april, 23:59.

1 Doelstelling

Het doel van dit practicum is om een compilertransformatie te schrijven die controleert of arrays binnen hun grenzen geadresseerd worden[1]. Hiervoor zullen we gebruik maken van de LLVM compilerinfrastructuur, een veelzijdige verzameling bibliotheken en toepassingen om het ontwikkelen van compilers te vereenvoudigen.

2 Opzetten van de werkomgeving

Aangezien de LLVM API snel evolueert, is het belangrijk dezelfde versie te gebruiken. Voor dit practicum moet je ontwikkelen tegen versie 3.5 van de API (of een *minor release* zoals 3.5.1). Om de nodige LLVM bibliotheken te installeren in de VM of op jouw PC[2], voer je volgende commando's uit in een terminalvenster:

```
cd ~/Downloads
wget http://users.elis.ugent.be/~tbesard/compilers/llvm-3.5.1.bin.tar.bz2
sudo tar -xvf llvm-3.5.1.bin.tar.bz2 -C /
```

De API documentatie die bij deze versie van LLVM hoort kan je raadplegen op `http://users.elis.ugent.be/~tbesard/compilers/llvm/`. Je kan ook de documentatie op de officiële website gebruiken, zie `http://llvm.org/docs/doxygen/html/`, maar weet dat die documentatie over de laatste nieuwe versie gaat en dus soms incompatibel kan zijn.

Als front-end compiler zal je gebruik maken van `cheetah`, een heel eenvoudige compiler die een minimale subset van C ondersteunt. De compiler genereert LLVM bitcode, die je door middel van `llc` compileert naar assembleertaal, en tenslotte met `gcc` tot een uitvoerbaar bestand converteert (dit omvat het assembleren en linken met de standaard C bibliotheek):

```
./cheetah test/$PROGRAM.c -o test/$PROGRAM.ll
llc -march=x86 test/$PROGRAM.ll -o test/helloworld.s
gcc -m32 test/$PROGRAM.s -o test/$PROGRAM.exe
```

[1] zie boek: p. 164 "A sermon on safety", p. 425 "Array-bounds checks"

[2] enkel Linux is ondersteund, en zorg in geval van een 64-bit OS dat je de nodige pakketten voor 32-bit compilatie (zoals `gcc-multilib`) geïnstalleerd hebt

```
# Alternatief, met pipes:
./cheetah test/$PROGRAM.c \
  | llc -march=x86 \
  | gcc -x assembler -m32 -o test/$PROGRAM.exe -
```

Je kan deze commando's manueel invoeren om zo volledige controle te hebben over het compilatieproces (let wel op dat je de LLVM binaries uit `/opt/llvm-3.5/bin` gebruikt), maar voor de meest gangbare sequenties kan je gebruik maken van de `Makefile`:

```
# Gegeven: test/$PROGRAM.c
make test/$PROGRAM.ll       # genereer tekstuele LLVM bitcode
make test/$PROGRAM.s        # genereer x86-assembly
make test/$PROGRAM.exe      # genereer een uitvoerbaar bestand
```

Je moet hierbij niet steeds manueel de drie verschillende `make` commando's uitvoeren, als je bijvoorbeeld `make test/helloworld.exe` uitvoert zal automatisch `helloworld.ll` en `helloworld.s` gegenereerd worden.

3 Introductie tot LLVM bitcode

De LLVM intermediaire representatie is een laag-niveau SSA representatie die zich goed leent tot analyses en transformaties. Een eenvoudige `for` lus wordt er voorgesteld zoals in Tabel 1.

Code is gegroepeerd in zogenaamde *basic blocks*, en controleverloop binnenin de applicatie verandert enkel op de grenzen van basic blocks. Elk basic block moet dan ook getermineerd worden met een instructie die het controleverloop stuurt (of gewoon door laat lopen door te branchen naar het volgende basic block); dergelijke instructies zijn allemaal afgeleid van de `llvm::TerminatorInst` superklasse.

Om de LLVM IR te wijzigen zal je steeds gebruik maken van de bijhorende API's. Zo heeft de `llvm::Value` klasse bijvoorbeeld een functie `replaceAllUsesWith`, of kan je een `llvm::IRBuilder` object aanmaken, die met de `setInsertionPoint` net voor een bestaande instructie richten, om dan nieuwe code in te voegen met een van de verschillende `Create` functies.

4 Geheugentoegangen in LLVM

Voor dit practicum gaan we aandacht besteden aan geheugentoegangen in arrays, en controleren of die binnen de correcte grenzen vallen. Zonder dergelijke beschermingsmaatregelen kan een programma crashen als het uitgevoerd wordt:

```
$ cat test/overflow.c
  int foo[10]; int n = 10;
  foo[n] = 0;
$ make test/overflow.exe && test/overflow.exe
  Segmentation fault
```

```
int n = 10;

int sum = 0;


for (
      int i = 0;


      i < n;


      i = i+1

    )

    {
      sum = sum + i;


    }
```

```
entry:
  %n = alloca i32
  store i32 10, i32* %n
  %sum = alloca i32
  store i32 0, i32* %sum
  br label %for.init

for.init:
  %i = alloca i32
  store i32 0, i32* %i
  br label %for.cond

for.cond
  %0 = load i32* %i
  %1 = load i32* %n
  %2 = icmp slt i32 %0, %1
  br i1 %2, label %for.body,
            label %for.end

for.inc:
  %6 = load i32* %i
  %7 = add i32 %6, 1
  store i32 %7, i32* %i
  br label %for.cond

for.body:
  %3 = load i32* %sum
  %4 = load i32* %i
  %5 = add i32 %3, %4
  store i32 %5, i32* %sum
  br label %for.inc

for.end:
  ret i32 0
```

Tabel 1: Een `for` lus met bijhorende LLVM bitcode.

De bedoeling van dit practicum is dat je dergelijke crashes voorkomt, en in plaats daarvan een bericht toont aan de gebruiker alvorens de uitvoering te beëindigen met een `SIGABRT`:

```
$ make test/overflow.exe && test/overflow.exe
  overflow.checked.exe: overflow.c:2: Assertion
                        'out-of-bounds array access' failed.
  Aborted
```

Als je naar de LLVM bitcode van het voorbeeldbestand `test/overflow.c` kijkt, zie je dat de geheugentoegang bestaat uit een `getelementptr` instructie om het adres van het element te berekenen, en een geheugenoperatie (hier een `store`) die de bewerking uitvoert:

```
%foo = alloca [10 x i32]
%0 = ...
%1 = getelementptr [10 x i32]* %foo, i32 0, i32 %0
store i32 0, i32* %1
```

5 Beschermen van geheugentoegangen

Er zijn verschillende mogelijkheden om bovenstaande geheugentoegangen te analyseren en beschermen. Voor dit practicum zal je de `getelementptr` instructie prefixen met instructies die de indices controleren.

De operanden van de `getelementptr` instructie zijn in ons geval als volgt: een basispointer die wijst naar waar de array opgeslagen is, een index die in termen van de basispointer indexeert[3], en een index die het element selecteert. Zie `http://llvm.org/docs/GetElementPtr.html` voor meer informatie.

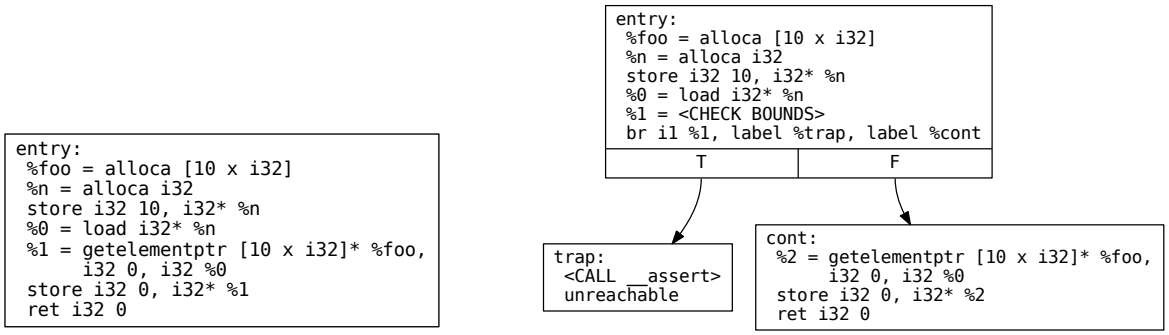
Uit deze operanden kan je afleiden wat de index van het opgevraagde geheugenelement in de array is. Merk op dat die index voorgesteld zal worden door een symbolisch `llvm::Value` object, dat niet noodzakelijk een bekende waarde heeft bij het compileren.

Als je de index van het opgevraagde element statisch (tijdens het compileren) kan controleren, dan moet de pass ook direct een foutbericht tonen door `llvm::report_fatal_error` aan te roepen. Indien je de index moet controleren tijdens het uitvoeren, genereer dan de nodige code en roep conditioneel de `__assert` functie aan (deel van de Sys V ABI), die een foutbericht toont en vervolgens de uitvoering van het programma afbreekt. In beide gevallen moet je de debug informatie die de `cheetah` front end genereert (dit zijn de `!dbg` metadata structuren in de bitcode) gebruiken om een nuttig foutbericht te genereren.

Als de controle tijdens het uitvoeren van het programma gebeurt, zal je het controleverloop moeten splitsen afhankelijk van het resultaat van het resultaat. Je kan dit visualiseren door de `opt` tool aan te roepen met de optie `-dot-cfg`:

```
make test/$PROGRAM.ll
opt -dot-cfg /test/$PROGRAM.ll
xdot cfg.$FUNCTIE.dot
```

[3] aangezien LLVM hier weet dat de pointer van het type `[10 x i32]` is, zal deze index steeds 0 zijn, of het geadresseerde geheugen zou zich *voorbij* de hele array bevinden



(a) Oorspronkelijke code (b) Met bounds checking

Figuur 1: Controleverloop voor en na toevoegen bounds checking.

Voor een enkele geheugentoegang krijg je zo het controleverloop in Figuur 1a (zonder bounds checking) en Figuur 1b (met bounds checking). Je kan ook de tool `llvm-diff` gebruiken om snel twee bitcode-bestanden te vergelijken.

6 Ontwikkelen van een LLVM pass

Om de hierboven beschreven beschermingen te implementeren, zal je een LLVM *function pass* implementeren. Dit is een stukje code dat voor elke functie in een objectbestand aangeroepen wordt, en waarbinnen je de functie kan analyseren, transformeren, of zelfs verwijderen.

In het archief van dit practicum vindt je reeds een bestand `BoundsCheck/Pass.cpp` die de basis zal vormen van een bounds checking pass. In zijn huidige vorm gaat de pass enkel op zoek naar alle `getelementptr` instructies, en worden er enkele objecten geïnitialiseerd die je nodig zal hebben doorheen het practicum.

Om de pass te compileren kan je opnieuw gebruik maken van de meegeleverde `Makefile`, waarna je met de `opt` tool de pass kan inladen en uitvoeren:

```
make BoundsCheck/libLLVMBoundsCheck.so

make test/$PROGRAM.ll
opt -load BoundsCheck/libLLVMBoundsCheck.so -cheetah-boundscheck \
    -o test/$PROGRAM.checked.bc test/$PROGRAM.ll
llvm-dis test/$PROGRAM.checked.bc -o test/$PROGRAM.checked.ll
```

Merk op dat `opt` steeds binaire bitcode wegschrijft (extensie `bc`), waardoor een aanroep naar `llvm-dis` nodig is om bitcode in tekstueel formaat te bekomen.

Je kan bovenstaande commando's opnieuw automatisch uitvoeren met behulp van `make`, door de bestandsextensie met `checked.` te prefixen. Hierbij wordt zelfs de function pass automatisch gehercompileerd, indien de bronbestanden ervan gewijzigd zijn:

```
# (Her)compileer de function pass indien nodig,
# en genereer de nodige .ll, .s en .exe bestanden
# met bounds checking ingeschakeld:
make test/$PROGRAM.checked.exe
```

7 Opgave

Concreet zal je dus de volgende aspecten moeten implementeren in `Pass.cpp`:

- Afleiden van array dimensies en indices van geheugentoegangen (indien mogelijk reeds bij het compileren, zoniet tijdens het uitvoeren).
- Controleren van de grenzen, en genereren van controleverloop in functie daarvan.
- Voorzien van failure basic block, waarbij je debug informatie gebruikt om een bijpassende foutmelding te voorzien.

Test elk van deze aspecten grondig: voeg test-cases toe aan de map `test/` opdat elke situatie geverifieerd wordt, en controleer het resulterende gedrag. Programmeer ook steeds defensief; gebruik bijvoorbeeld `assert` statements wanneer je bepaalde assumpties neemt.

Tenslotte moet je ook een analyse maken (in `VERSLAG.txt`) van de kost van bounds checking: is die (statistisch) significant, hoe verklaar je dat (in functie van de gegenereerde code en de onderliggende hardware), en hoe denk je eventueel de kost te kunnen verkleinen?