Detection of Non-Size Increasing Programs in Compilers
Implementation of Implicit Complexity Analysis

Jean-Yves Moyen\textsuperscript{1}  Thomas Rubiano\textsuperscript{2}

\textsuperscript{1}Department of Computer Science
University of Copenhagen (DIKU)
Supported by the Marie Curie action “Walgo” program H2020-MSCA-IF-2014, number 655222

\textsuperscript{2}Laboratoire Informatique Paris Nord
Université Paris 13 & University of Copenhagen (DIKU)
PhD funded by the ELICA ANR project (ANR-14-CE25-0005)

6 avril 2016
Introduction

- ICC deals with syntactic criterion that guarantee some property (complexity bounds)
- A lot of theories:
  - Bounded Recursion (A. Cobham)
  - Safe/Normal Recursion (S. Bellantoni and S. Cook)
  - Size-change and termination (C.S. Lee, N.D. Jones and A.M. Ben-Amram), Quasi-interpretation and verification of resources (J.Y. Marion, R. Amadio, G. Bonfante, J.Y. Moyen, R. Péchoux), Polynomes MWP (L. Kristiansen and N.D. Jones)
  - Non-Size-Increasing programs (M. Hofmann)
  - ...
Most of them concern “toy languages”
20 years of ICC’s theories: time to fill the gap between theories and actual programs
But real languages are complex...
A good language level: Intermediate Representations
A good start: Detection of NSI Programs
Motivations 2/2

Compilers developers mainly focus on optimizations. . .

- Analysis and Optimizations are not so far apart
- Providing proven bounds on space and time: a safety and a security property
Motivations 2/2

Compilers developers mainly focus on optimizations. . .

- Analysis and Optimizations are not so far apart
- Providing proven bounds on space and time : a safety and a security property

A proof of concept to show that ICC and Compilers can fuel each other
Section 1

NSI Programs
First idea of safe recursion from S. Bellantoni and S. Cook: repeated iteration is a source of exponential growth.

The study of Non Size Increasing was introduced by M. Hofmann: “it’s not harmful to iterate function which does not increase the size of its data.”

We want to detect and to certify that a program computes (or can compute) within a constant amount of space.
Hofmann detects non size increasing programs by adding a special type 🟦 which can be seen as the type of pointers to free memory.

Example (insertion without 🟦)

\[
\begin{align*}
\text{insert}( & \ y, \ [] \) \rightarrow \ \text{cons}( \ y, \ [] \\
\text{insert}( & \ y, \ \text{cons}( \ x, \ xs) \) \rightarrow \\
& \quad \text{if} \ x < y \\
& \quad \text{then} \ \text{cons}( \ x, \ (\text{insert}( \ y, \ xs))) \\
& \quad \text{else} \ \text{cons}( \ y, \ \text{cons}( \ x, \ xs))
\end{align*}
\]
Hofmann detects non size increasing programs by adding a special type $\Diamond$ which can be seen as the type of **pointers** to free memory.

**Example (insertion with $\Diamond$)**

\[
\text{insert}(d, y, []) \rightarrow \text{cons}(d, y, []) \\
\text{insert}(d, y, \text{cons}(d', x, xs)) \rightarrow \\
\quad \text{if } x < y \\
\quad \text{then } \text{cons}(d', x, (\text{insert}(d, y, xs))) \\
\quad \text{else } \text{cons}(d, y, \text{cons}(d', x, xs))
\]

simply, the constructor consumes one diamond $d : \Diamond$ then exponentiation is not possible anymore.
Insert represented as **CFG**
(Control Flow Graph is a graph composed of **basic blocks** composed of **basic instructions**):
Add a **weight** (corresponding to the space used by the program) to the CFG and we obtain the following **RCG** (Resource Control Graph):

\[
\begin{align*}
&l = [] \quad l \neq [] \\
&x \geq y \\
&x < y \\
&+1
\end{align*}
\]
Section 2

Compilers and Intermediate Representation
Principles

Optimizer

Compiler
Principles

Frontend

C

C++

Java

IR

Optimizer

Compiler
Principles

- C++
- C
- Java

Intermediate Representation (IR)

Optimizer

Compiler

Frontend

Backend

- X86
- ARM
- MIPS
Principles

Compiler

Frontend

IR

Optimizer

IR

Backend

C

C++

Java

X86

ARM

MIPS
Principles

The diagram illustrates the flow of compilation for different programming languages to various architectures. It shows the interaction between Frontend, Optimizer, Analysis, Compiler, and Backend stages. Languages like C, C++, Java, etc., are processed through an Optimizer, which then feeds into Analysis, leading to the Compiler for specific architectures such as X86, ARM, MIPS, etc.
To make some optimizations we need analysis

These optimizations and analysis are managed as passes on the programs’ Intermediate Representation (Gimple/RTL for GCC, LLVM IR for LLVM)

A lot of passes already exist. For instance in gcc:

```
$ gcc -c --help=optimizers -Q | wc -l
184
$ gcc -c -O --help=optimizers -Q | grep enabled | wc -l
76
$ gcc -c -O2 --help=optimizers -Q | grep enabled | wc -l
105
$ gcc -c -O3 --help=optimizers -Q | grep enabled | wc -l
112
```
A lot of passes already used by default:

```
$ gcc -fdump-tree-all -fdump-rtl-all loop.c -o loopgcc
$ ll loop.c.*
loop.c.001t.tu
loop.c.003t.original
loop.c.004t.gimple
loop.c.006t.vcg
...
loop.c.150r.expand
loop.c.151r.sibling
loop.c.153r.initvals
loop.c.154r.unshare
...
$ ll loop.c.* | wc -l
43
```

A pass-manager stores data in memory from analysis made previously for next ones.
Order is given as argument to the **pass manager**:

```bash
$ llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments
Pass Arguments: -targetlibinfo -no-aa -tbaa -scoped-noalias -assumption-tracker
               -basicaa -notti -verify-di -ipsccp -globalopt -deadargelim -domtree
               -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -inline
               -functionattrs -argpromotion -sroa -domtree -early-cse -lazy-value-info
               -jump-threading -correlated-propagation -simplifycfg -domtree -instcombine
               -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa
               -loop-rotate -licm -loop-unswitch -instcombine -scalar-evolution
               -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -function_tti
               -loop-unroll -memdep -mldst-motion -domtree -memdep -gvn -memdep -memcpyoct
               -sccp -domtree -instcombine -lazy-value-info -jump-threading
               -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -domtree
               -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob
               -block-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution
               -slp-vectorizer -simplifycfg -domtree -instcombine -loops -loop-simplify
               -lcssa -scalar-evolution -function_tti -loop-unroll
               -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge
               -verify -verify-di
```

A lot of passes are used to prepare optimizations or clean the IR. (e.g. detection of $\sum_{i=1}^{n} i$ is made by finding specific pattern)
<table>
<thead>
<tr>
<th>Category</th>
<th>GCC</th>
<th>LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>= (+)</td>
<td>=</td>
</tr>
<tr>
<td>Popular</td>
<td>high</td>
<td>(deb)</td>
</tr>
<tr>
<td>Old</td>
<td>28 years</td>
<td>12 years</td>
</tr>
<tr>
<td>Licensing</td>
<td>GPLv3</td>
<td>University of Illinois/NCSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Open Source License (no copyleft)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(and Tools)</td>
</tr>
<tr>
<td>Modular</td>
<td>(−)?</td>
<td>built for</td>
</tr>
<tr>
<td>Documentation</td>
<td>(−)?</td>
<td>+</td>
</tr>
<tr>
<td>Community</td>
<td>?</td>
<td>Huge and active!</td>
</tr>
<tr>
<td>Contributions</td>
<td>(2012) 16 commits/day, 470 devs, 7.3 Mlines</td>
<td>(2014) 34 commits/day, 2.6 Mlines</td>
</tr>
</tbody>
</table>
**LLVM Tools**

- LLVM framework comes with lot of tools to compile and optimize code:

<table>
<thead>
<tr>
<th>FileCheck</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileUpdate</td>
<td>diagtool</td>
</tr>
<tr>
<td>arcmt-test</td>
<td>fpcmp</td>
</tr>
<tr>
<td>bugpoint</td>
<td>lic</td>
</tr>
<tr>
<td>c-arcmt-test</td>
<td>lli</td>
</tr>
<tr>
<td>c-index-test</td>
<td>lli-child-target</td>
</tr>
<tr>
<td>llvm-PerfectSf</td>
<td>llvm-mc</td>
</tr>
<tr>
<td>llvm-ar</td>
<td>llvm-mcmarkup</td>
</tr>
<tr>
<td>llvm-as</td>
<td>llvm-nm</td>
</tr>
<tr>
<td>clang-check</td>
<td>llvm-bc analyzer</td>
</tr>
<tr>
<td>clang-format</td>
<td>llvm-c-test</td>
</tr>
<tr>
<td>clang-modernize</td>
<td>llvm-config</td>
</tr>
<tr>
<td>clang-tblgen</td>
<td>llvm-cov</td>
</tr>
<tr>
<td></td>
<td>llvm-dis</td>
</tr>
<tr>
<td></td>
<td>llvm-dwarf dump</td>
</tr>
<tr>
<td></td>
<td>llvm-extract</td>
</tr>
<tr>
<td></td>
<td>llvm-link</td>
</tr>
<tr>
<td></td>
<td>llvm-lit</td>
</tr>
<tr>
<td></td>
<td>llvm-lto</td>
</tr>
<tr>
<td></td>
<td>obj2yaml</td>
</tr>
<tr>
<td></td>
<td>opt</td>
</tr>
<tr>
<td></td>
<td>pp-trace</td>
</tr>
<tr>
<td></td>
<td>llvm-objdump</td>
</tr>
<tr>
<td></td>
<td>llvm-ranlib</td>
</tr>
<tr>
<td></td>
<td>llvm-readobj</td>
</tr>
<tr>
<td></td>
<td>llvm-rtdyld</td>
</tr>
<tr>
<td></td>
<td>llvm-stress</td>
</tr>
<tr>
<td></td>
<td>llvm-symbolizer</td>
</tr>
<tr>
<td></td>
<td>llvm-tblgen</td>
</tr>
<tr>
<td></td>
<td>macho-dump</td>
</tr>
<tr>
<td></td>
<td>modularize</td>
</tr>
<tr>
<td></td>
<td>clang</td>
</tr>
<tr>
<td></td>
<td>clang++</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>llvm-size</td>
</tr>
<tr>
<td></td>
<td>rm-cstr-calls</td>
</tr>
<tr>
<td></td>
<td>tool-template</td>
</tr>
<tr>
<td></td>
<td>yaml2obj</td>
</tr>
</tbody>
</table>
LLVM Tools

- LLVM framework comes with lot of tools to compile and optimize code:
  - FileCheck
  - FileUpdate
  - arcmt-test
  - bugpoint
  - c-arcmt-test
  - c-index-test
  - llvm-PerfectSf
  - llvm-ar
  - llvm-as
  - clang-check
  - clang-format
  - clang-modernize
  - clang-tblgen
  - count
  - diagtool
  - fpcmp
  - llc
  - lli
  - lli-child-target
  - llvm-mc
  - llvm-mcmarkups
  - llvm-nm
  - llvm-bcanalyzer
  - llvm-c-test
  - llvm-config
  - llvm-cov
  - llvm-dis
  - llvm-dwarfdump
  - llvm-extract
  - llvm-link
  - llvm-lit
  - llvm-lto
  - obj2yaml
  - opt
  - pp-trace
  - llvm-objdump
  - llvm-ranlib
  - llvm-readobj
  - llvm-rtdyld
  - llvm-stress
  - llvm-symbolizer
  - llvm-tblgen
  - macho-dump
  - modularize
  - clang
  - clang++
  - not
  - llvm-size
  - rm-cstr-calls
  - tool-template
  - yaml2obj

- LLVM offers good structures and tools to easily navigate and manage Instructions

- Create a module with a pass is pretty simple
LLVM Intermediate Representation

LLVM-IR is a **Typed Assembly Language (TAL)** and a **Static Single Assignment (SSA)** based representation. This provides:

- type safety
- low-level operations
- flexibility
- capability to represent high-level languages “cleanly”

An IR is **source-language-independent**, then optimizations and analysis should work on every languages (properly translated to this IR).
Instruction set

LLVM-IR has a RISC-like instruction set:

<table>
<thead>
<tr>
<th>Terminator</th>
<th>Bin Operator</th>
<th>Bitwise Operator</th>
<th>Stack and addressing</th>
<th>other</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>add</td>
<td>shl/r</td>
<td>alloca</td>
<td>phi</td>
<td>…</td>
</tr>
<tr>
<td>br</td>
<td>sub</td>
<td>and</td>
<td>load</td>
<td>select</td>
<td>…</td>
</tr>
<tr>
<td>switch</td>
<td>mul</td>
<td>or</td>
<td>store</td>
<td>call</td>
<td>…</td>
</tr>
<tr>
<td>invoke</td>
<td>div</td>
<td>xor</td>
<td>getelementptr</td>
<td>icmp</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Focus on the **call** instruction able to call **libc allocation function** (**free** and **malloc**).
We go over LLVM data structures through iterators:

- **Iterator over a Module**
  - gives a list of Function

- **Iterator over a Function**
  - gives a list of BasicBlock

- **Iterator over a Basic Block**
  - gives a list of Instruction

- **Iterator over an Instruction**
  - gives a list of Operands

```c
//iterate on each module’s functions
for(Module_Iterator F=M.begin(), Fe=M.end();
   F!=Fe; ++F){
  //iterate on each function’s basic block
  for(Function_Iterator b=F.begin(),
      be=F.end(); b!=be; ++b){
    //iterate on each BB’s instructions
    for(BasicBlock_Iterator I=b->begin(),
        ie=b->end(); I!=ie; ++I){
      ...
    }
  }
}
```
Section 3

Our analysis, Demos and Conclusions
In our case we want to build a RCG and find the heaviest path regarding to allocation memory.

- LLVM tools already provide the CFG\(^1\)...
- We can compute the weight of each **Basic Block** by counting number of allocation on...

---

1. Recall : A CFG starts with one *entry-block* and has several *exit-blocks*, that builds the structured programming concept
Bellman-Ford’s Algorithm

we can calculate the heaviest path and detect positive loops with the Bellman-Ford’s Algorithm

1. Initialization:
   set all vertices to minus infinite weight except the first one

2. Relaxation of each vertices starting from the first one:
   take the highest weight regarding to all the edges converging toward this node

3. Check for positive-weight cycle:
   if one edge \( u \rightarrow v \) with a weight \( w \) has
   \[ \text{weight}[u] + w > \text{weight}[v] \] it’s a positive cycle
Is the program NSI?

This analysis just provide an answer to the question “Is the program/function NSI?”. We consider all positives loops as occurred a non-determined number of time.
Conclusion

- We built a static analyzer in almost 200 lines of code thanks to the modularity of the compiler.
- It can be seen as two passes: the first one build a RCG (reusable) and the second detect positive loops.
- tested on reverse, concat, insertion sort and quick sort.
A lot of work remains to be done

- find dependence between each source file
- every libraries used should have been analyzed before
- customizing standard dynamic allocations and deallocation
- approximate a *space complexity* and maybe the *termination*