Detection of Non-Size Increasing Programs in Compilers

Implementation of Implicit Complexity Analysis

Jean-Yves Moyen[1] Thomas Rubiano[2]

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

[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)
  - ...

Motivations 1/2

- 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

Bounding Complexity

- 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

NSI and Imperative programs

- 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 without $\Diamond$)

```
insert(   y, []) -> cons(   y, [])
insert(   y, cons(   x, xs)) ->
        if x<y
        then cons(   x, (insert(   y, xs)))
        else cons(   y, cons(   x, xs))
```

NSI and Imperative programs

- 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 $\Diamond$)

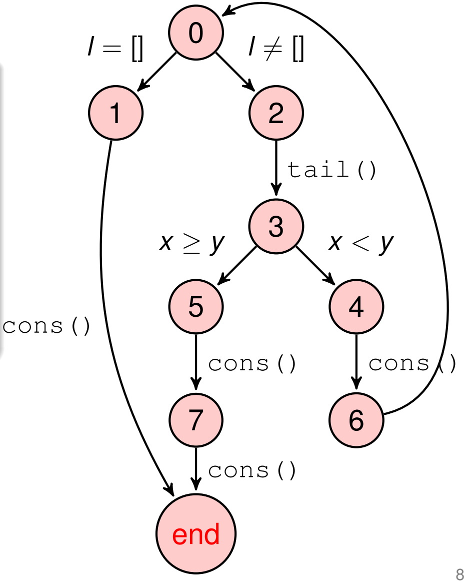
```
insert(d, y, []) -> cons(d, y, [])
insert(d, y, cons(d', x, xs)) ->
        if x<y
        then cons(d', x, (insert(d, y, xs)))
        else cons(d, y, cons(d', x, xs))
```

- simply, the constructor consumes one diamond $d : \Diamond$ then exponentiation is not possible anymore

CFG view

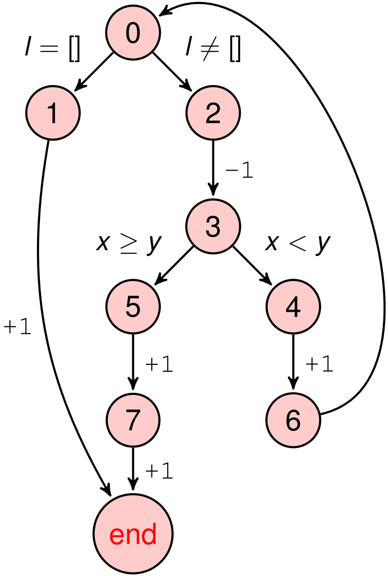
```
insert(d, y, []) -> cons(d,
    y, [])
insert(d, y, cons(d', x,
    xs)) ->
        if x<y
        then cons(d', x,
             (insert(d, y,
             xs)))
        else cons(d, y,
             cons(d', x, xs))
```

Insert represented as **CFG** (Control Flow Graph is a graph composed of **basic blocks** composed of **basic instructions**) :



Analogy with Space-RCG

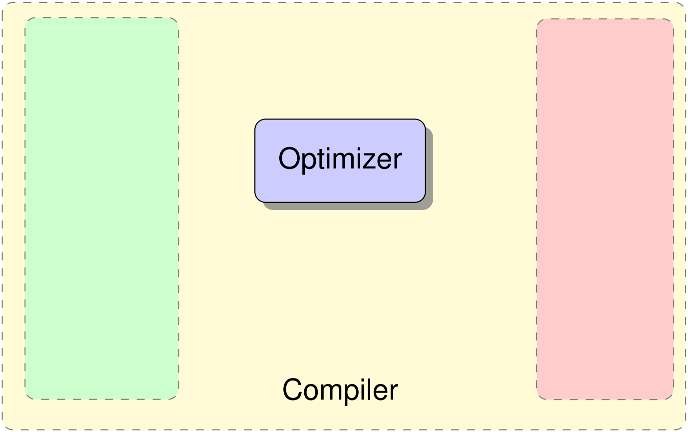
Add a **weight** (corresponding to the space used by the program) to the CFG and we obtain the following **RCG** (Resource Control Graph) :



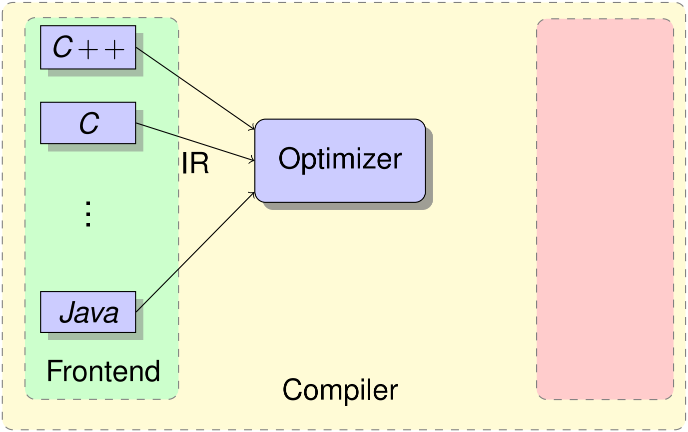
Section 2

Compilers and Intermediate Representation

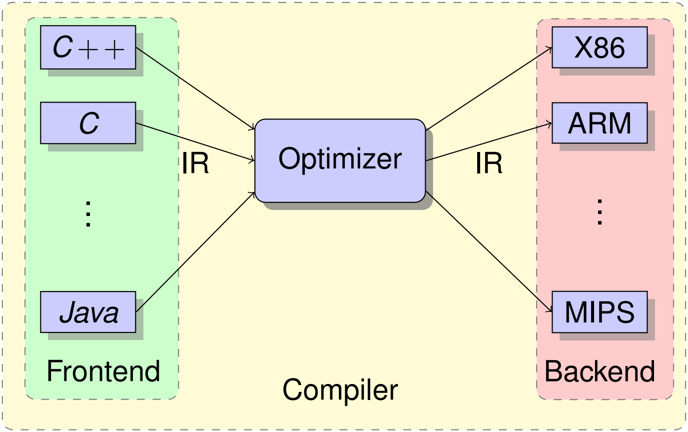
Principles



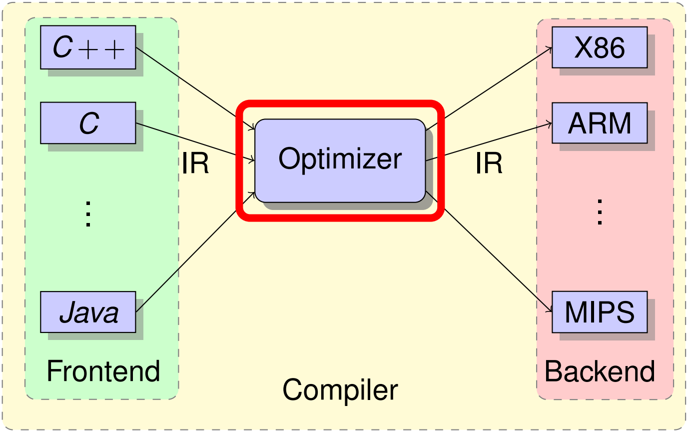
Principles



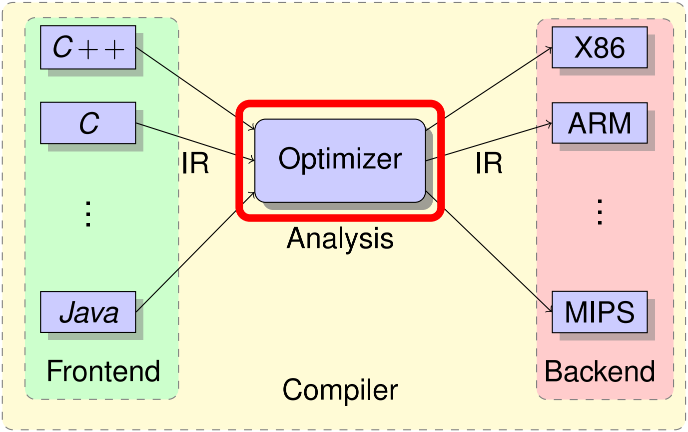
Principles


C++
C
⋮
Java
Frontend
IR
Optimizer
IR
X86
ARM
⋮
MIPS
Backend
Compiler

Principles


C++
C
Java
Frontend
IR
Optimizer
IR
X86
ARM
MIPS
Backend
Compiler

Principles


C++
C
⋮
Java
Frontend
IR
Optimizer
Analysis
IR
X86
ARM
⋮
MIPS
Backend
Compiler

Analysis

- 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
```

Analysis

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
```

Gimple

RTL

A pass-manager stores data in memory from analysis made previously for next ones.

Order

Order is given as argument to the **pass manager** :

```
$ 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 -memcpyopt
     -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)

GCC and LLVM (and Compcert)

	GCC	LLVM
Performance	$= (+)$	$=$
Popular	high	$\nearrow$ (deb)
Old	28 years	12 years
Licensing	GPLv3	University of Illinois/NCSA Open Source License (no copyleft) (and Tools)
Modular	$(-)$?	built for
Documentation	$(-)$?	$+$
Community	?	Huge and active !
Contributions	(2012) 16 commits/day, 470 devs, 7.3 Mlines	(2014) 34 commits/day, 2.6 Mlines

LLVM Tools

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

FileCheck	count	llvm-dis	llvm-stress
FileUpdate	diagtool	llvm-dwarfdump	llvm-symbolizer
arcmt-test	fpcmp	llvm-extract	llvm-tblgen
bugpoint	llc	llvm-link	macho-dump
c-arcmt-test	lli	llvm-lit	modularize
c-index-test	lli-child-target	llvm-lto	clang
llvm-PerfectSf	llvm-mc	obj2yaml	clang++
llvm-ar	llvm-mcmarkup	opt	not
llvm-as	llvm-nm	pp-trace	llvm-size
clang-check	llvm-bcanalyzer	llvm-objdump	rm-cstr-calls
clang-format	llvm-c-test	llvm-ranlib	tool-template
clang-modernize	llvm-config	llvm-readobj	yaml2obj
clang-tblgen	llvm-cov	llvm-rtdyld	

LLVM Tools

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

FileCheck	count	llvm-dis	llvm-stress
FileUpdate	diagtool	llvm-dwarfdump	llvm-symbolizer
arcmt-test	fpcmp	llvm-extract	llvm-tblgen
bugpoint	llc	llvm-link	macho-dump
c-arcmt-test	lli	llvm-lit	modularize
c-index-test	lli-child-target	llvm-lto	clang
llvm-PerfectSf	llvm-mc	obj2yaml	clang++
llvm-ar	llvm-mcmarkup	opt	not
llvm-as	llvm-nm	pp-trace	llvm-size
clang-check	llvm-bcanalyzer	llvm-objdump	rm-cstr-calls
clang-format	llvm-c-test	llvm-ranlib	tool-template
clang-modernize	llvm-config	llvm-readobj	yaml2obj
clang-tblgen	llvm-cov	llvm-rtdyld	

- 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 :

Terminator	Bin Operator	Bitwise Operator	Stack and addressing	other	...
ret	add	shl/r	alloca	phi	
br	sub	and	load	select	
switch	mul	or	store	call	
invoke	div	xor	getelementptr	icmp	
...	...	...	...	...	

Focus on the `call` instruction able to call `libc` allocation function (`free` and `malloc`).

IR in memory

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

```
//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

Building RCG

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
   $weight[u] + w > weight[v]$ 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*