Non Size Increasing programs and Compilers

Implementation of Implicit Complexity

Thomas Rubiano

PhD supervised by V. Mogbil & J.-Y. Moyen,
in collaboration with V. Danjean,
funded by the Elica Project

Motivations 1/2

- ICC helps to predict and control resources
- A lot of theories :
  - 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)
  - ...

Motivations 2/2

- After 20 years of ICC's theories, maybe it's time to go in compilers ?
- But it's complicated
- NSI Programs analysis seems to be a good start

Introduction 1/3

- We want to detect and to certify that a program computes (or can compute) within a constant amount of space
- The study of Non Size Increasing was introduced by M. Hofmann
  First idea (safe recursion from S. Bellantoni and S. Cook) : restrict iterations of functions
- An example is to allow only iterations on non size increasing functions

Introduction 2/3

- Hofmann detects non size increasing programs in a typed functional language by adding a special type $\Diamond$ which can be seen as the type of pointers to free memory

Example (reverse without $\Diamond$)

```
(* with accumulator *)

rev(l) -> rev1(l,nil)
rev1(nil,acc) -> acc
rev1(cons(  h,t),acc) -> rev1(t,cons(  h,acc))
```

Introduction 2/3

- Hofmann detects non size increasing programs in a typed functional language by adding a special type $\diamondsuit$ which can be seen as the type of pointers to free memory

Example (reverse with $\diamondsuit$)

```
(* with accumulator *)

rev(l) -> rev1(l,nil)
rev1(nil,acc) -> acc
rev1(cons(d,h,t),acc) -> rev1(t,cons(d,h,acc))
```

- simply, the constructor consumes one diamond d

Introduction 2/3

- Hofmann detects non size increasing programs in a typed functional language by adding a special type $\diamondsuit$ which can be seen as the type of pointers to free memory

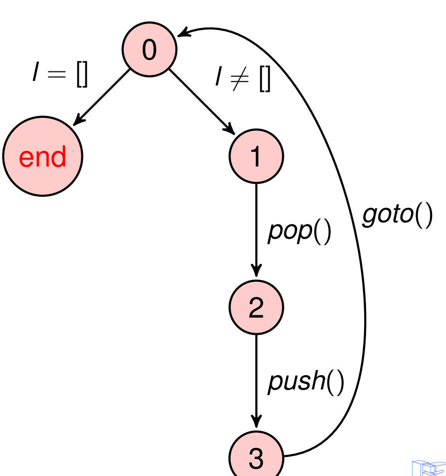
Example (reverse on Stack-Machine)

```
0: if l = [] then goto 4;
1: h,d:=pop(l); (*frees a d*)
2: push(d,h,acc); (*consumes a d*)
3: goto 0;
4: end;
```

Introduction 3/3

```
0: if l = [] then goto 4;
1: h,d:=pop(l);
2: push(d,h,acc);
3: goto 0;
4: end;
```

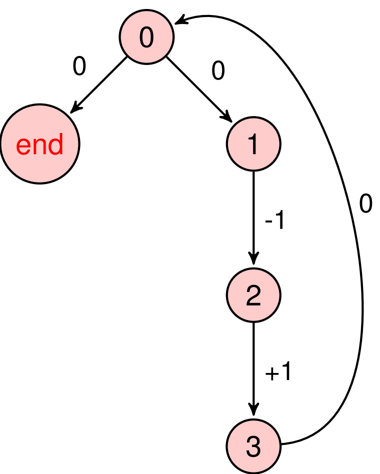
Reverse represented as CFG :



Analogy with Space-RCG

```
0: if l = [] then goto 4;
1: h,d:=pop(l);
2: push(d,h,acc);
3: goto 0;
4: end;
```

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



Section 2

Compilers

Principles

Optimizer

Compiler

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 uses analysis made previously to select the next passes

GCC and LLVM

	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 : OPT/PassManager and LLVM IR

- 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 : OPT/PassManager and LLVM IR

- 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-mcmarkup
  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

Section 3

Data structures, a Graph issue and demos

Intermediate Representation

IR looks like assembly language but it’s more readable…

IR Data Structure

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 a Instruction gives a list of Operands

```
//iterate on each module's functions
for(Moduleiterator F=M.begin(),
      Fe=M.end(); F!=Fe; ++F){
  //iterate on each function's basic block
  for(Functioniterator b=F.begin(),
        be=F.end(); b!=be; ++b){
    //iterate on each BB's instructions
    for(BasicBlockiterator I=b->begin(),
          ie=b->end(); I!=ie; ++I){
      ...
```

Bellman-Ford's Algorithm

- In our case we want to build a RCG and find the heaviest path.

Bellman-Ford's Algorithm

- In our case we want to build a RCG and find the heaviest path.
- We already have the CFG...

Bellman-Ford's Algorithm

- In our case we want to build a RCG and find the heaviest path.
- We already have the CFG. . .
- We can find the weight of each BasicBlock. . .

Bellman-Ford's Algorithm

- In our case we want to build a RCG and find the heaviest path.
- We already have the CFG. . .
- We can find the weight of each BasicBlock. . .
- we can calculate the heaviest path. . .

Bellman-Ford's Algorithm

- In our case we want to build a RCG and find the heaviest path.
- We already have the CFG...
- We can find the weight of each BasicBlock...
- we can calculate the heaviest path...
- and detect positive loops with the Bellman-Ford's Algorithm

Bellman-Ford's Algorithm

1. Initialization :
   all vertices with -infinite weight except the first
2. Relaxation of each vertices :
   take the highest weight regarding 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

A new issue

This is easy in one source file...
But if we consider the fact that we need all the function's weight, we will need to develop a tool capable to find dependences between each source file and collect all the informations needed to calculate the local functions.

AMADIO (R.), COUPET-GRIMAL (S.), ZILIO (S. Dal) and JAKUBIEC (L.). –
A functional scenario for bytecode verification of resource bounds. *In : Computer Science Logic, 12th International Workshop, CSL'04*. pp. 265–279. –
Springer.

BAILLOT (P.) and TERUI (K.). –
Light types for polynomial time computation in lambda calculus. *Information and Computation*, vol. 201 (1), 2009, pp. 41–62.

BELLANTONI (S.) and COOK (S.). –
A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, vol. 2, 1992, pp. 97–110.

BONFANTE (G.), MARION (J.-Y.) and MOYEN (J.-Y.). –

Quasi-interpretations a way to control resources. *Theoretical Computer Science*, vol. 412 (25), 2011, pp. 2776 – 2796.

GIRARD (J.-Y.). –
Linear Logic. *Theoretical Computer Science*, vol. 50, 1987, pp. 1–102.

HOFMANN (M.). –
Linear types and Non-Size Increasing polynomial time computation. *In : Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pp. 464–473.

LEE (C. S.), JONES (N. D.) and BEN-AMRAM (A. M.). –
The Size-Change Principle for Program Termination. pp. 81–92. –
ACM press.