Implementation of Implicit Complexity

Midterm defense

Thomas Rubiano

PhD supervised by V. Mogbil in Université Paris 13 &
J. G. Simonsen in Kobenhavn Universitet &
J.-Y. Moyen,
funded by the Elica Project

Introduction

- ICC helps to predict and control resources
- A lot of theories :
  - Safe/Normal Recursion (S. Bellantoni and S. Cook)
  - Size-change and termination (C.S. Lee, N.D. Jones and A.M. Ben-Amram)
  - 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. . .

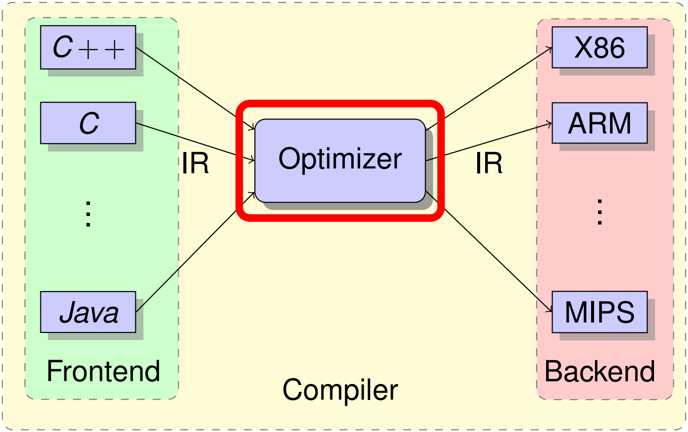
- Analysis and Optimizations are not so far apart
- Providing proven bounds on space and time : a safety and a security property

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

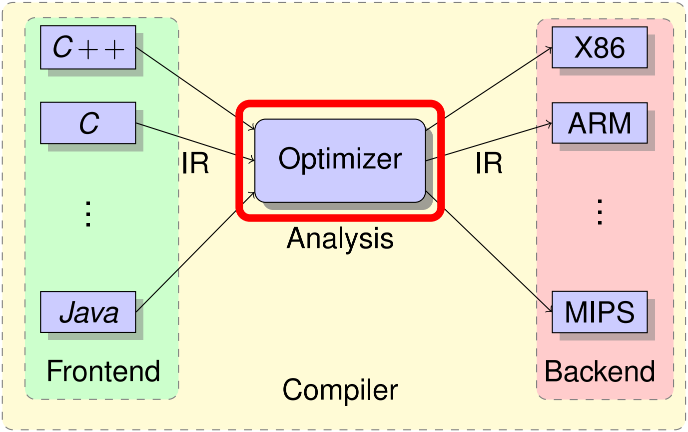
Section 1

Compilers

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

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      } Gimple
loop.c.004t.gimple
loop.c.006t.vcg
...
loop.c.150r.expand
loop.c.151r.sibling       } RTL
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

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

	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 Intermediate Representation

- LLVM-IR is a **Typed Assembly Language** (TAL) and a **Static Single Assignment** (SSA) based representation. This provides :
- An IR is **source-language-independent**, then optimizations and analysis should work on every languages (properly translated to this IR).

Section 2

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 is 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 $\diamondsuit$ which can be seen as the type of **pointers to free memory** in Imperative Programs.

Example (insertion without $\diamondsuit$)

```
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** in Imperative Programs.

Example (insertion with $\diamond$)

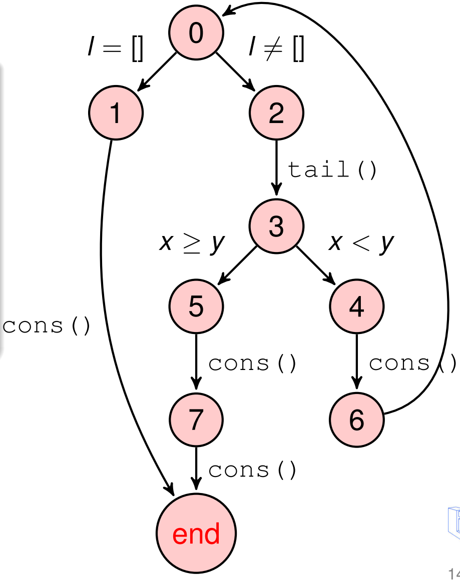
```
insert(d, y, []) -> cons(d, y, [])
insert(d, y, cons(d', x, xs)) ->
        if x<y
        then cons(d', x, (insert(d, y, xs)))
        else cons(d, y, cons(d', x, xs))
```

- simply, the constructor consumes one diamond d then exponentiation is not possible anymore

CFG view

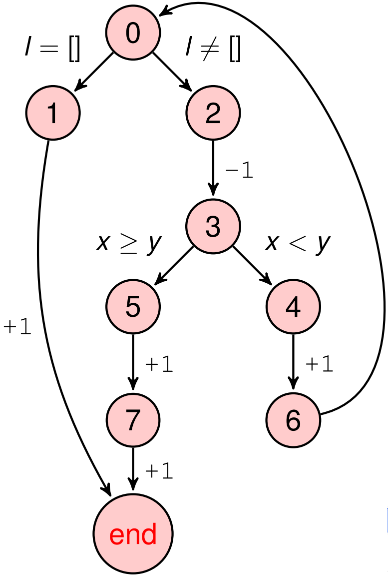
```
insert(d, y, []) -> cons(d,
     y, [])
insert(d, y, cons(d', x,
     xs)) ->
         if x<y
         then cons(d', x,
              (insert(d, y,
              xs)))
         else cons(d, y,
              cons(d', x, xs))
```

Insert represented as **CFG** (Control Flow Graph) :



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) :



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

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

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.
- available on **github here**

Section 3

Quasi-invariant block code motion

From ICC techniques to compiler optimization ?

- From an idea of Lars Kristiansen, about language theory and proof on semantic equivalence after an optimization
- Interesting techniques of data flow analysis in "*mwp*-bounds" and in termination analysis using "size-change graphs"
- could help to trace and gather dependencies between variables : build a dependency graph
- What if we try to do so for compilers optimizations ?

Motivations

- Learn about variables dependencies around loops
- Learn about loop optimizations, especially loop-invariant detection and hoisting
- Provide another point of view and maybe a new optimization : "*Quasi-invariant block code motion*"
- In a way to assist programmers
- Seems to not be implemented in compilers. . . (not in LLVM, maybe in GCC. . .)

Quasi-Invariants

- A quasi-invariant is a variable which does not change after a certain number of loop execution
- A degree of invariance is the number of time we need to compute the loop until the variable is stable
- It could be very long for a human. . .

```
while(i<100){
    z=y*y; //2
    use(z);
    y=x+x; //1
    use(y);
    i=i+1;
}
```

Matrix

Definition

This Data Flow Graph can be represented as a matrix $N \times N$ with $N = |var(\mathtt{C})|$, we will note C the corresponding matrix to $\mathtt{C}$.

$\mathtt{C} := [x_0 = x_0 + 1; x_2 = 0];$

$$C = \begin{bmatrix} 1 & \emptyset & \emptyset \\ \emptyset & 0 & \emptyset \\ \emptyset & \emptyset & \emptyset \end{bmatrix}$$

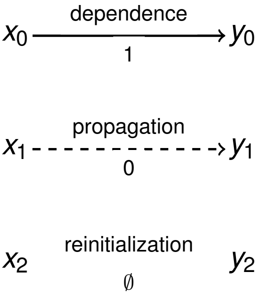


FIGURE – Matrix of dependence

Chunks

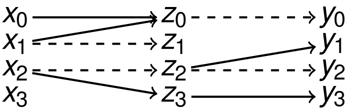
- Command Composition
- See one block as one command
- Hoist an entire block (could be a loop !)

Multipath and Composition example

Example of the following sequence :

$C_1 := [x_0 = x_0 + x_1; x_3 = x_2 + 2];$
$C_2 := [x_1 = x_2; x_3 = x_3 * 2];$

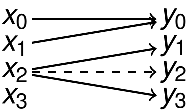
C_1 $\quad$ C_2



$$C_1 = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & 0 & \emptyset & \emptyset \\ \emptyset & \emptyset & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix} C_2 = \begin{bmatrix} 0 & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & 1 & 0 & \emptyset \\ \emptyset & \emptyset & \emptyset & 1 \end{bmatrix}$$

(a) Multipaths

$[C_1; C_2]$



$$C_1 \bullet C_2 = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & \emptyset & \emptyset & \emptyset \\ \emptyset & 1 & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

(b) Composition

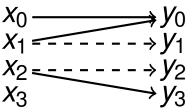
Condition example

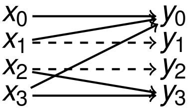
Example of the following sequence : C := if E then C_1; with

E $:= x_3 \geq 0$

$C_1 := [x_0 = x_0 + x_1; x_3 = x_2 + 2];$

E | C_1 | [if E then C_1]



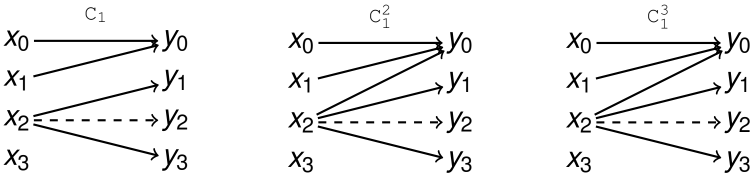


$$E = \begin{bmatrix} \emptyset \\ \emptyset \\ \emptyset \\ 1 \end{bmatrix} \quad C_1 = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & 0 & \emptyset & \emptyset \\ \emptyset & \emptyset & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix} \quad (C_1)^{\mathtt{E}} = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & 0 & \emptyset & \emptyset \\ \emptyset & \emptyset & 0 & 1 \\ 1 & \emptyset & \emptyset & 1 \end{bmatrix}$$

FIGURE – Condition

Loop `while` example

Example of the following sequence (`C`$_1$ is the composition presented previously) : `C := while E do C`$_1$; with `E` $:= x_3 \geq 0$
`C`$_1$ $:= [x_0 = x_0 + x_1; x_3 = x_2 + 2; x_1 = x_2; x_3 = x_3 * 2];$



$$C_1 = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & \emptyset & \emptyset & \emptyset \\ \emptyset & 1 & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix} \quad C_1^2 = \mathcal{C}_1^2 = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & \emptyset & \emptyset & \emptyset \\ 1 & 1 & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix} \quad C_1^3 = \mathcal{C}_1^* = \begin{bmatrix} 1 & \emptyset & \emptyset & \emptyset \\ 1 & \emptyset & \emptyset & \emptyset \\ 1 & 1 & 0 & 1 \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

FIGURE – Finding fix point of dependence (simple example)

Loop `while`

Let `C` be a command such as : `C` := `while E do` $\mathtt{C}_1$;.

- first occurrence of $\mathtt{C}_1$ will influence the second one and so on
- we consider the number of iteration undecidable
- Let's define $\mathcal{C}^* = limit(\mathcal{C}^k)$.
- $C_{i,j} = \bigoplus_k (E_i \oplus (\mathcal{C}_1^*)_{k,j})$ or we can simplify the notation as $C = (\mathcal{C}_1^*)^{\mathrm{E}}$

Matrix Algebra

The Matrix representing a *DFG* is composed of elements in $\mathcal{E} = \{\emptyset, 0, 1\}$. The elements in $\mathcal{E}$ are ordered as follows : $\emptyset < 0 < 1$. And we can introduce two operations noted $\oplus$ and $\otimes$ defined as below :

$\oplus_{max}$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	0	1
0	0	0	1
1	1	1	1

$\otimes_{+}$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	$\emptyset$	0	1
1	$\emptyset$	1	1

$\oplus$ could be seen as a *max* and $\otimes$ as a $+$ if we consider $\emptyset$ as $-\infty$.

Then the composition of matrices is computed as :
$C_{i,j} = \bigoplus_k (A_{i,k} \otimes B_{k,j})$ we can write $C = A \bullet B$.

Mutual independence of chunks

Definition

If $\texttt{C}_2$ *independent of* $\texttt{C}_1$ *and* $\texttt{C}_1$ *independent of* $\texttt{C}_2$ *then* $\texttt{C}_2$ *and* $\texttt{C}_1$ *are mutually independents :*

$$\texttt{C}_1 \asymp \texttt{C}_2$$

Example of the following sequence :
$\texttt{C}_1 := [x_0 = x_0 + x_1;$
$\texttt{C}_2 := [x_3 = x_2 + x_3 * 2];$

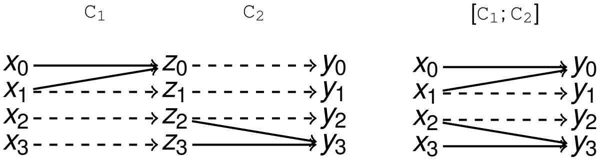


FIGURE – Composition of mutually independent chunks of commands

In this example, $\texttt{C}_1 \asymp \texttt{C}_2$ but $[\texttt{C}_1; \texttt{C}_2]$ is not self-independent.

Moving Independent Chunks

Lemma

Swapping commands (or chunks of commands) : If $\mathtt{C}_1 \asymp \mathtt{C}_2$ *then*

$$\mathtt{C}_1 ; \mathtt{C}_2 \equiv \mathtt{C}_2 ; \mathtt{C}_1$$

Lemma

Moving mutual independent commands out of `while` *: If* $\mathtt{C}_1 \asymp \mathtt{C}_2$ *and* $\mathtt{C}_1 \asymp \mathtt{C}_1$ *then*

$$\mathtt{while\ E\ do}\ [\mathtt{C}_1 ; \mathtt{C}_2] \equiv [\mathtt{if\ E\ then\ C}_1 ; \mathtt{while\ E\ do\ C}_2]$$

Figure out the invariance degree

Let suppose we have computed the list of dependencies for all commands. How to compute the degree of one command ?

1. Initialize every degrees to 0
2. Initialize the current command degree `cd` to ∞
3. **IF** there is no dependence for the current chunk return 1
4. **ELSE** for each dependence compute the degree `dd` of the command
    1. **IF** `cd <= dd` and the current command dominates this dependence **THEN** `cd = dd + 1`
       ELSE `cd = dd`

A toy for testing

- to validate, we implemented on a toy parser in python
- around 400 lines
- if you have some in mind ?

Last example

```
srand(time(NULL));
int n=rand()%100;
int j=0;
while(j<100){
    fact=1;
    i=1;
    while (i<n) {
        fact=fact*i;
        i=i+1;
    }
    j=j+1;
    use(fact);
}
```

```
srand(time(NULL));
int n = rand() % 100;
int j = 0;
if (j < 100)
{
  fact = 1;
  i = 1;
  while (i < n)
  {
    fact = fact * i;
    i = i + 1;
  }
  j = j + 1;
  use(fact);
}
while (j < 100)
{
  j = j + 1;
  use(fact);
}
```

Revelations !

- I discovered a paper few weeks ago...
  "A Loop Optimization Technique Based on Quasi-Invariance" by Litong Song, Yoshihiko Futamura, Robert Glück, Zhenjiang Hu - 2000
- We still have new concepts : Chunks, Compositions and type of dependencies

Questions !

Asked to the french community of compilation :

- Do you think it's relevant to write a pass on it ?
- Do you think it's relevant to write a paper on it ?

CV

- Conferences
  - Workshop DICE2016 Eindhoven (NSI programs)
  - French Community Of Compilation Aussois (Quasi-invariant block motion QIBM)
- Papers
  - DICE2016 (NSI programs)
  - draft for CC2017 (deadline 1st Nov) on (QIBM) or EuroLLVM2017
- Talks
  - DIKU Copenhagen (Compiler and IR introductions)
  - ELICA Bologna (QIBM)
- Courses
  - Summer school OPLSS2015 Eugene (2w)
  - Summer school + project CEMRACS2016 Luminy (6w)
  - Master Course (Complexity and Computation) University of Copenhagen (4m) Validated

- 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.