Implicit Computational Complexity meets Compilers

funded by ELICA (ANR-14-CE25-0005)

Thomas Rubiano

Jury :

Guillaume Bonfante	Université de Lorraine	Rapporteur
Christophe Fouqueré	Université Paris13	Examinateur
Laure Gonnord	Université Lyon1	Rapporteuse
Tobias Grosser	ETH Zürich	Examinateur
Stefano Guerrini	Université Paris13	Examinateur
Virgile Mogbil	Université Paris13	Directeur
Torben Mogensen	Københavns Universitet	Examinateur
Jean-Yves Moyen	Université Paris13	Encadrant
Ulrich Schöpp	Ludwig-Maximilians-Universität München	Examinateur
Jakob Grue Simonsen	Københavns Universitet	Directeur

What?

The recipe analogy

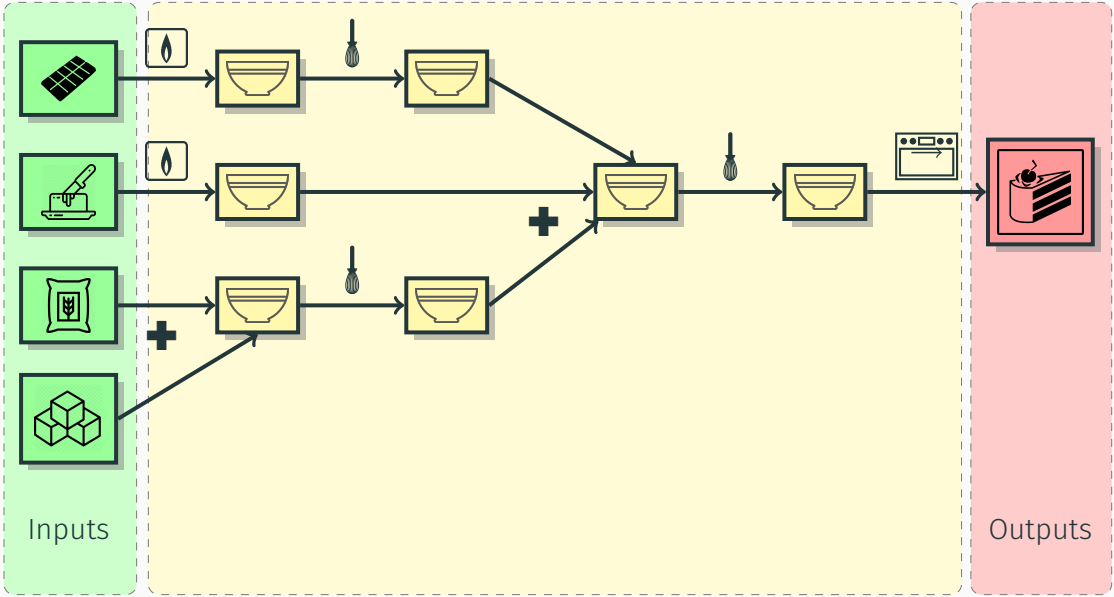
(the cake is a lie!)

Inputs

Outputs

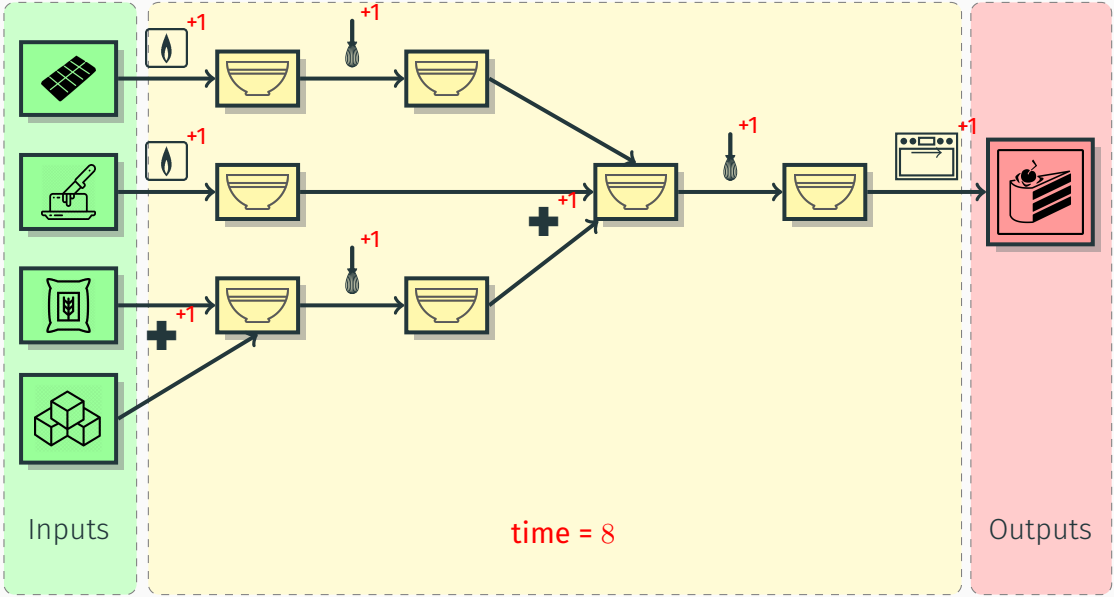
The recipe analogy

(the cake is a lie!)



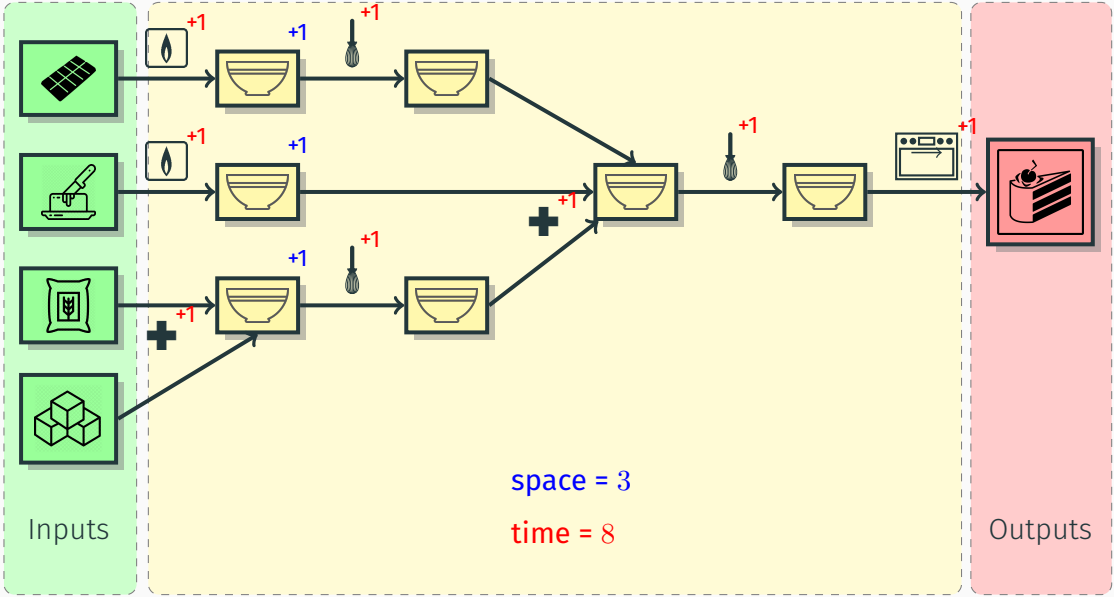
The recipe analogy

(the cake is a lie!)



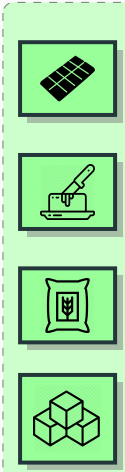
The recipe analogy

(the cake is a lie!)



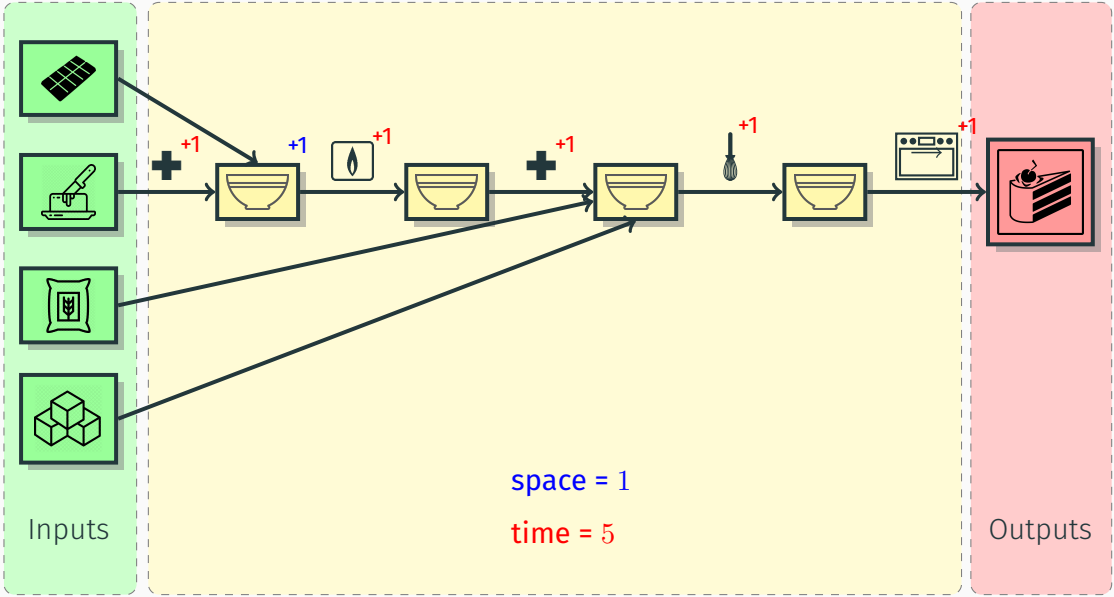
The recipe analogy

(the cake is a lie!)



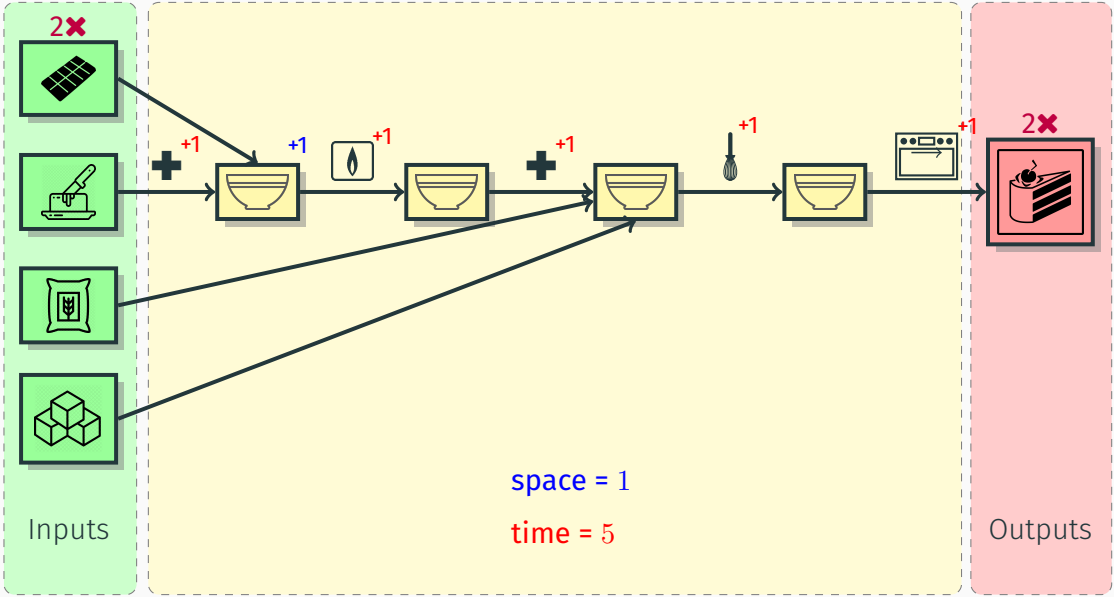
The recipe analogy

(the cake is a lie!)



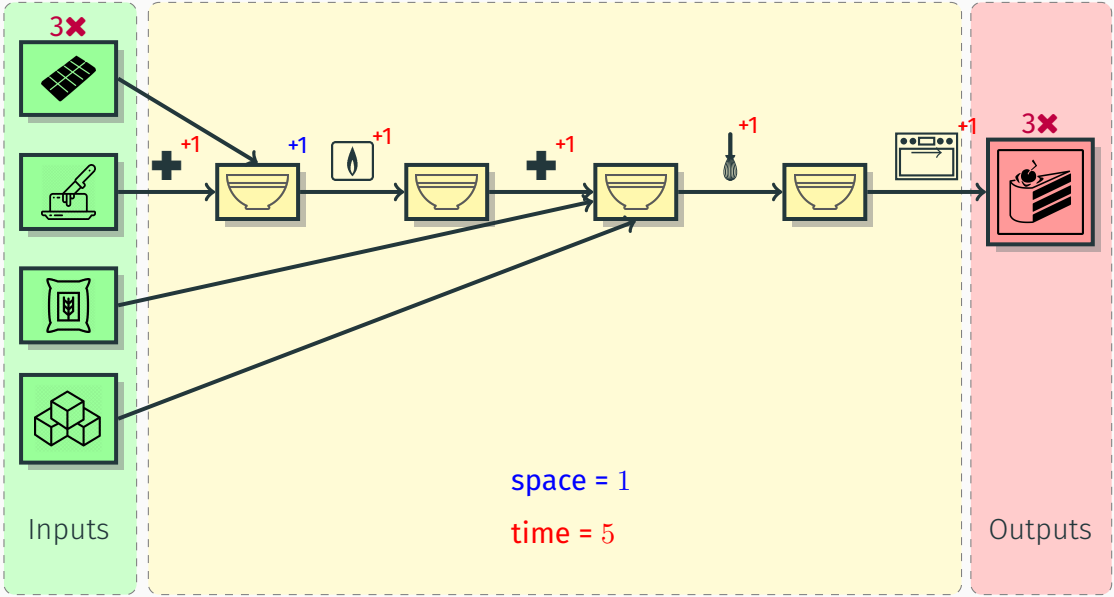
The recipe analogy

(the cake is a lie!)



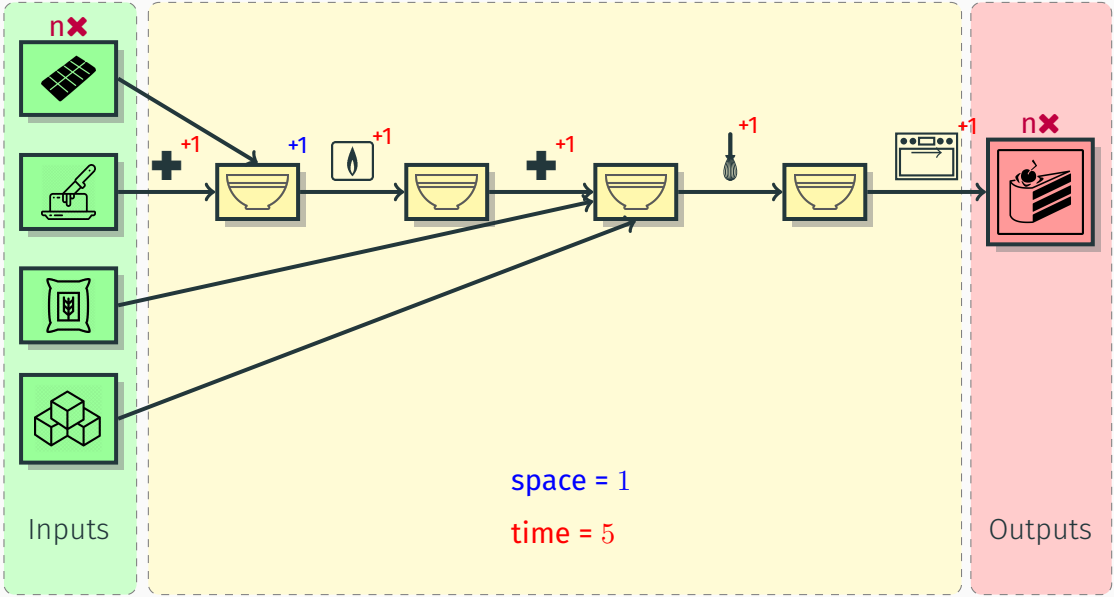
The recipe analogy

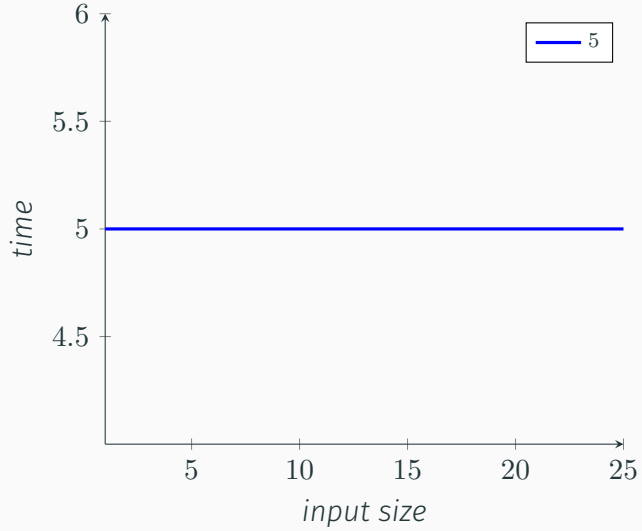
(the cake is a lie!)



The recipe analogy

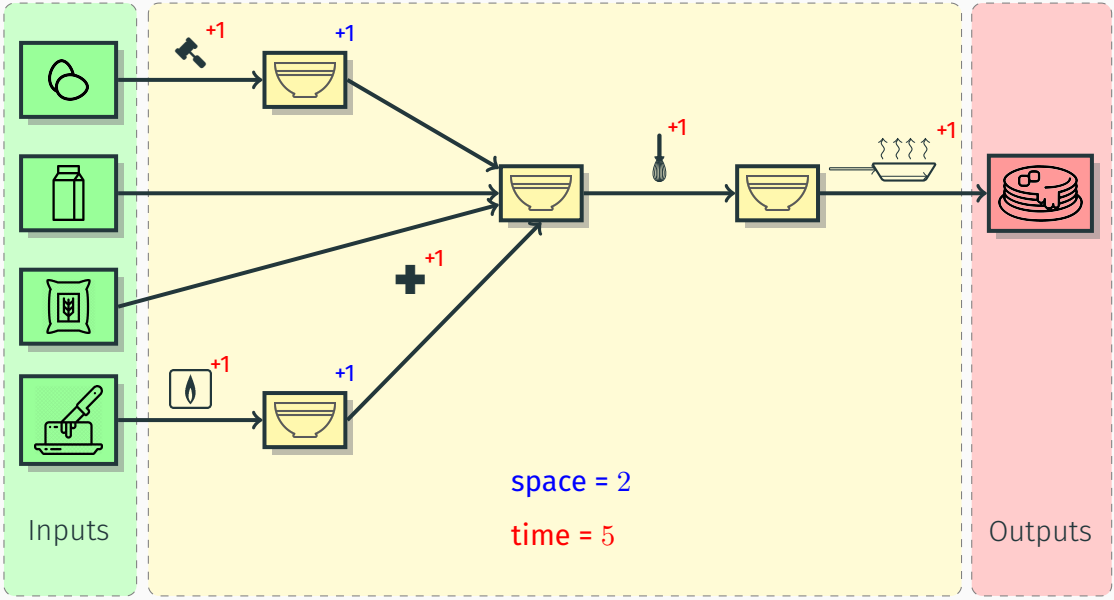
(the cake is a lie!)




6
5.5
5
4.5
time
5
10
15
20
25
input size
5

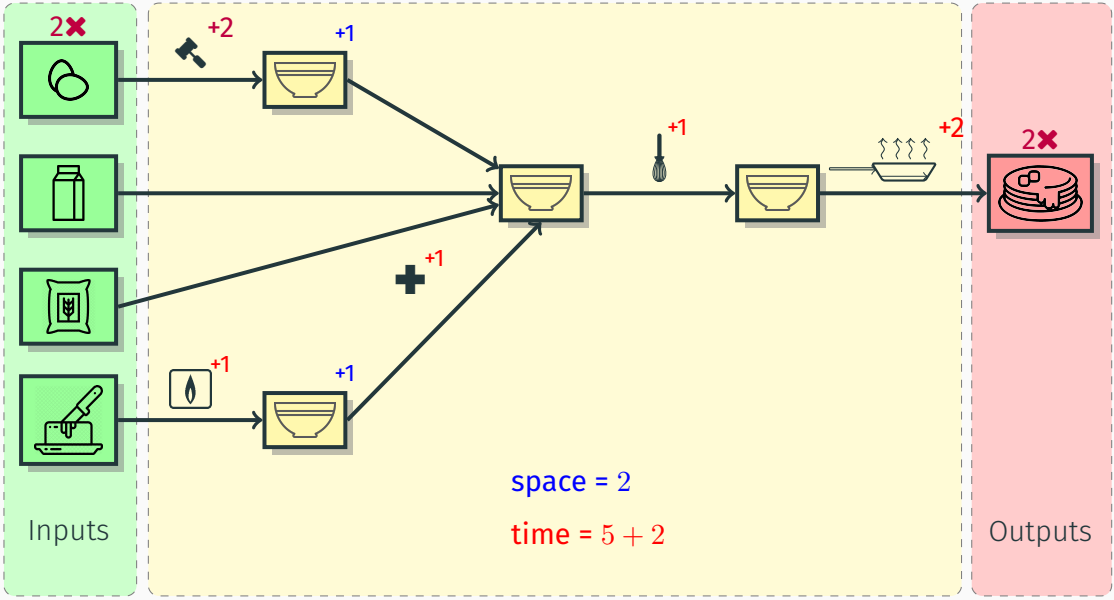
The recipe analogy (2)

(the crêpe is not a lie!)



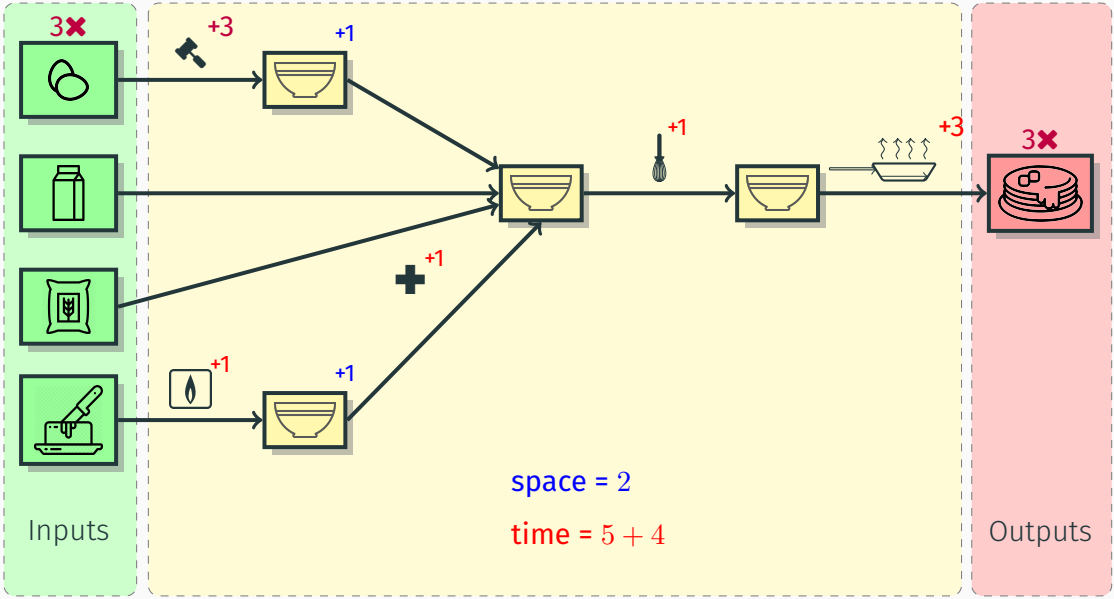
The recipe analogy (2)

(the crêpe is not a lie!)



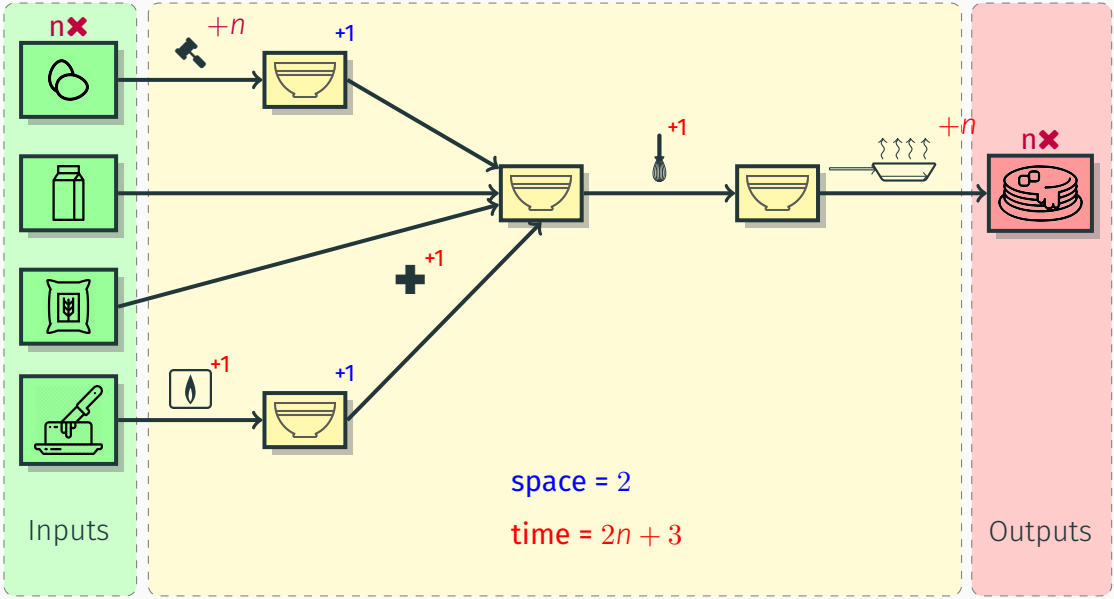
The recipe analogy (2)

(the crêpe is not a lie!)



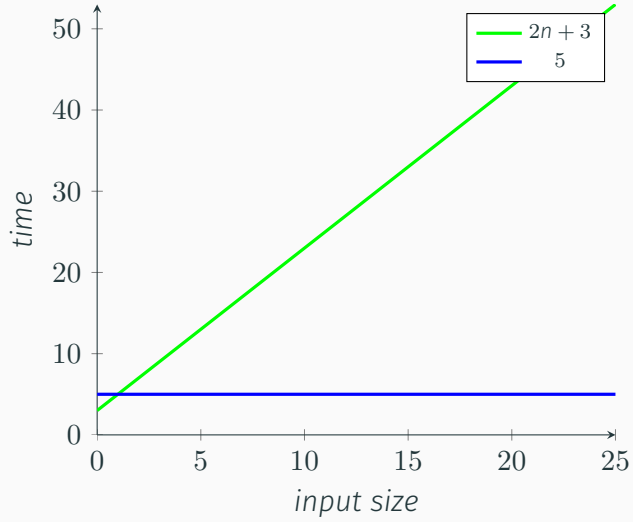
The recipe analogy (2)

(the crêpe is not a lie!)



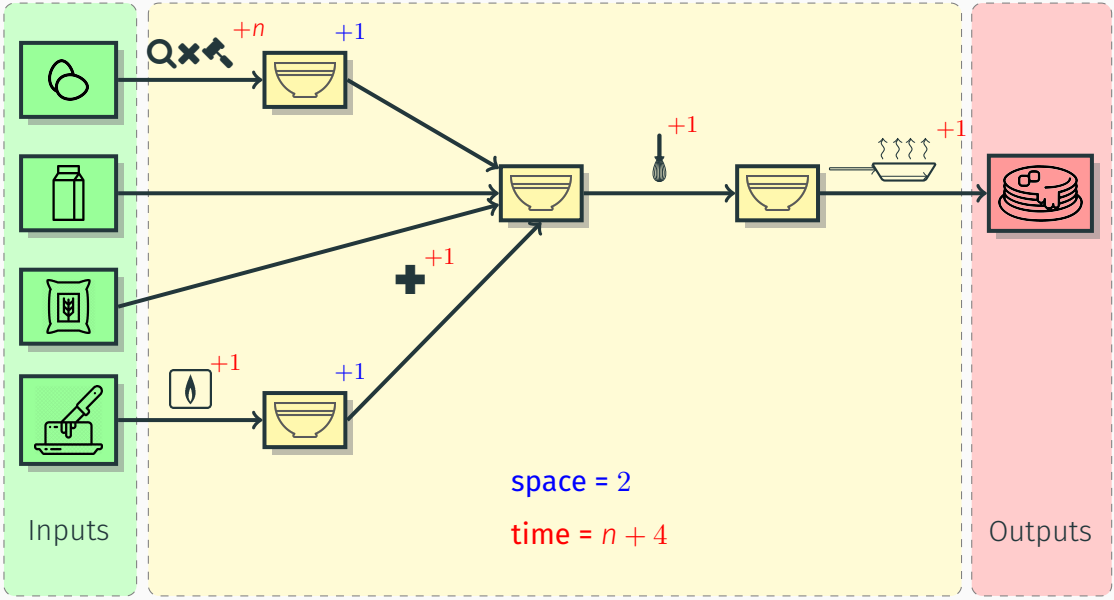
The recipe analogy (2)

(the crêpe is not a lie!)



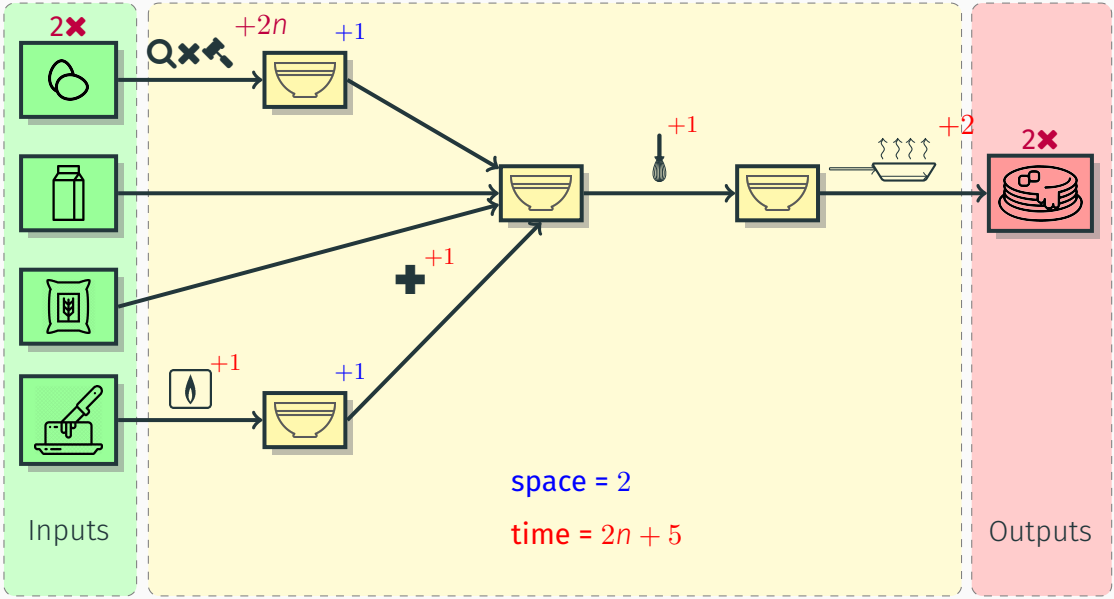
The recipe analogy (3)

(the crêpe is not a lie!)



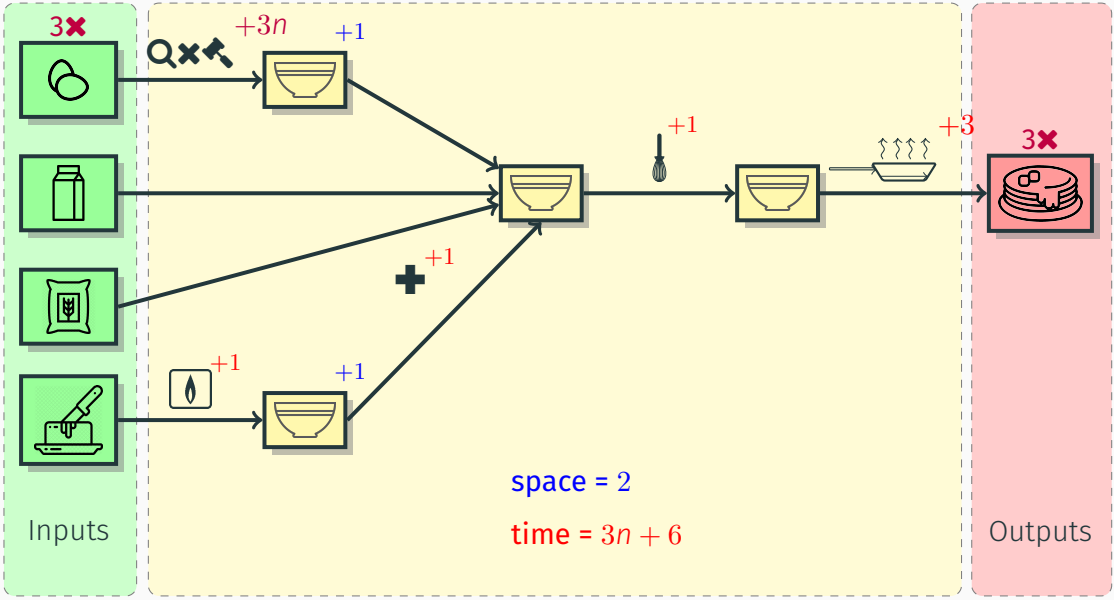
The recipe analogy (3)

(the crêpe is not a lie!)



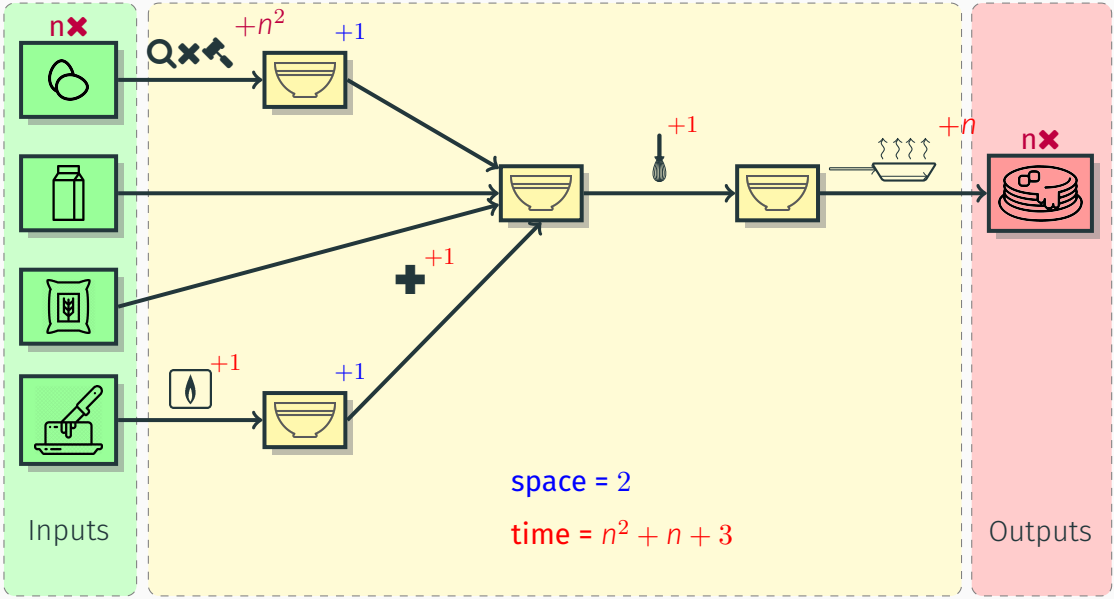
The recipe analogy (3)

(the crêpe is not a lie!)



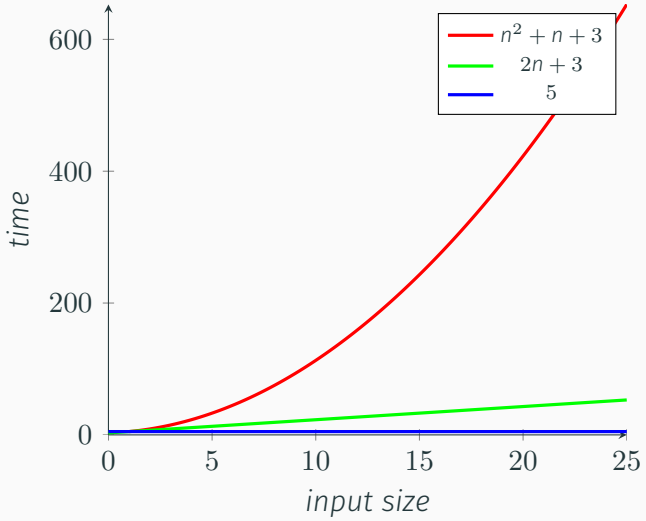
The recipe analogy (3)

(the crêpe is not a lie!)

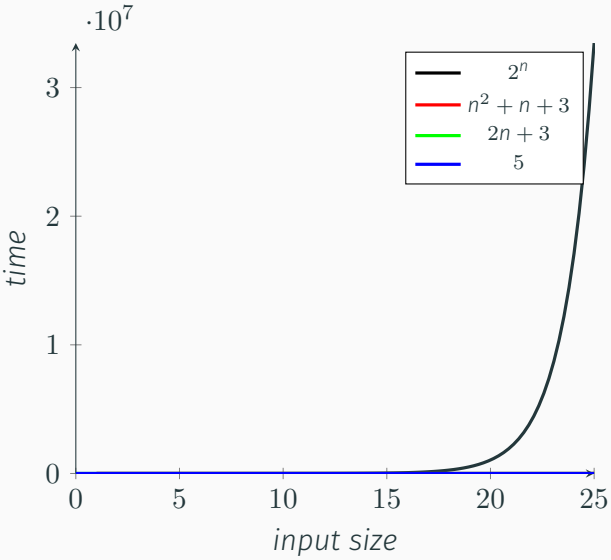


The recipe analogy (3)

(the crêpe is not a lie!)



“Exponential is bad”



Back to machines!

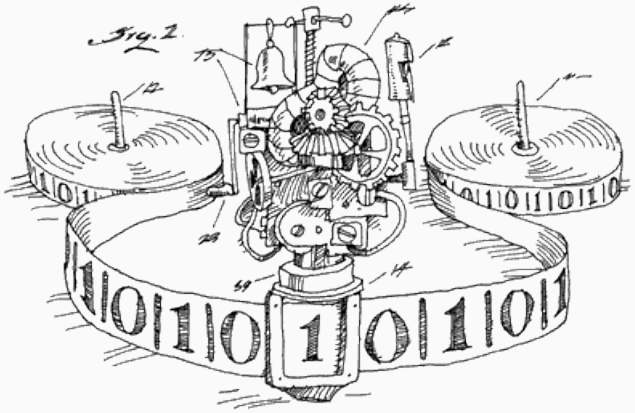


Fig. 2.
1 0 1 1 0 1 0 1 1 0 1

```
0: If reading 0, go to 3.
1: If reading 1, go to 10.
3: Write B.
4: Move right.
5: If reading 0, go to 4.
6: If reading 1, go to 4.
7: Move left.
8: If reading 0, go to 18.
9: Reject.
10: Write B.
11: Move right.
12: If reading 0, go to 11.
13: If reading 1, go to 11.
14: Move left.
15: If reading 1, go to 18.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right
22: Go to 0.
```

```
def is_palindrome(m):
    for i in range(0,int(len(m)/2)):
        if m[i]!=m[-(i+1)]:
            return False
    return True
```

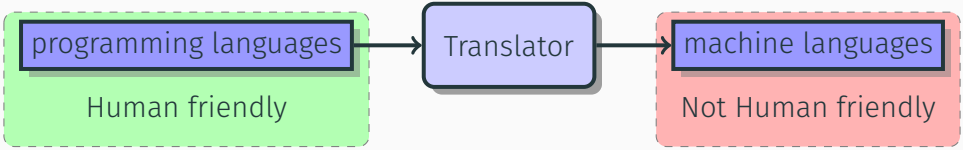
```
def is_palindrome(m):
    for i in range(0,int(len(m)/2)):
        if m[i]!=m[-(i+1)]:
            return False
    return True
```

Need a translator...

Compiler = Translator

```
def is_palindrome(m):
    for i in range(0,int(len(m)/2)):
        if m[i]!=m[-(i+1)]:
            return False
    return True
```

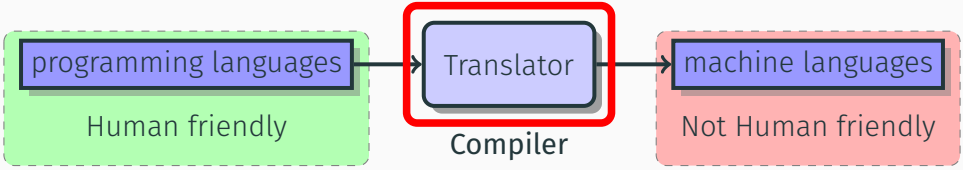
```
 0: If reading 0, go to 3.
 1: If reading 1, go to 10.
 3: Write B.
 4: Move right.
 5: If reading 0, go to 4.
 6: If reading 1, go to 4.
 7: Move left.
 8: If reading 0, go to 18.
 9: Reject.
10: Write B.
11: Move right.
12: If reading 0, go to 11.
13: If reading 1, go to 11.
14: Move left.
15: If reading 1, go to 18.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right
22: Go to 0.
```



Compiler = Translator

```
def is_palindrome(m):
    for i in range(0,int(len(m)/2)):
        if m[i]!=m[-(i+1)]:
            return False
    return True
```

```
0: If reading 0, go to 3.
1: If reading 1, go to 10.
3: Write B.
4: Move right.
5: If reading 0, go to 4.
6: If reading 1, go to 4.
7: Move left.
8: If reading 0, go to 18.
9: Reject.
10: Write B.
11: Move right.
12: If reading 0, go to 11.
13: If reading 1, go to 11.
14: Move left.
15: If reading 1, go to 18.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right
22: Go to 0.
```



Why ?

Warranty

GNU GPL license :

15. Disclaimer of Warranty.

> THERE IS NO WARRANTY FOR THE PROGRAM […] THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Warranty

GNU GPL license :

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM […] THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

What about Testing ?

Warranty

GNU GPL license :

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM […] THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

What about Testing ?

Warranty needs proofs !

Hard to prove something...

Halting problem is **undecidable** *over Turing machines*. (Alan Turing 1936)

Hard to prove something...

Halting problem is **undecidable** *over Turing machines*. (Alan Turing 1936)

But it is possible **in some cases**.

Completeness and Soundness

To the questions *“does the program have this property ?”* a :

- **Complete** analysis answers *“yes”* or *“no”*.
- **Sound** analysis answers *“yes”* or *“I don’t know”*.

Implicit Computational Complexity works

ICC helps to predict and control resources with syntactic criterion.
ICC's theories :

- Bounded Recursion (A. Cobham 1965)
- Safe/Normal Recursion (S. Bellantoni and S. Cook 1992)
- Size-change and termination (C.S. Lee, N.D. Jones and A.M. Ben-Amram 2001), Quasi-interpretation and verification of resources (J.Y. Marion, R. Amadio, G. Bonfante, J.Y. Moyen, R. Péchoux 2003), Polynomes MWP (L. Kristiansen and N.D. Jones 2011)
- Non-Size-Increasing programs (M. Hofmann 1999)
- ...

Implicit Computational Complexity works

ICC helps to predict and control resources with syntactic criterion.
ICC's theories :

- Bounded Recursion (A. Cobham 1965)
- Safe/Normal Recursion (S. Bellantoni and S. Cook 1992)
- Size-change and termination (C.S. Lee, N.D. Jones and A.M. Ben-Amram 2001), Quasi-interpretation and verification of resources (J.Y. Marion, R. Amadio, G. Bonfante, J.Y. Moyen, R. Péchoux 2003), Polynomes MWP (L. Kristiansen and N.D. Jones 2011)
- Non-Size-Increasing programs (M. Hofmann 1999)
- ...

Restricted or **"toy"** languages.

Motivations and contributions

Motivations

- 20 years of ICC : time to welcome those theories in **real** ~~world~~ **languages**.
- Give an accurate idea of the **real expressivity** of the ICC's methods.
- **Encourage ICC** community to bring their tools into real world languages.
- Contribute to the **"Complexity-certifying Compiler" dream**.

Motivations

- 20 years of ICC : time to welcome those theories in **real** ~~world~~ **languages**.
- Give an accurate idea of the **real expressivity** of the ICC's methods.
- **Encourage ICC** community to bring their tools into real world languages.
- Contribute to the **"Complexity-certifying Compiler" dream**.
- **Analysis and optimisations** are not so far apart...

Contributions

2 implementations of ICC methods in the LLVM compiler.

Contributions

2 implementations of ICC methods in the LLVM compiler.

Propose a new **Data-Flow Approach** for Complexity Analysis.

Contributions

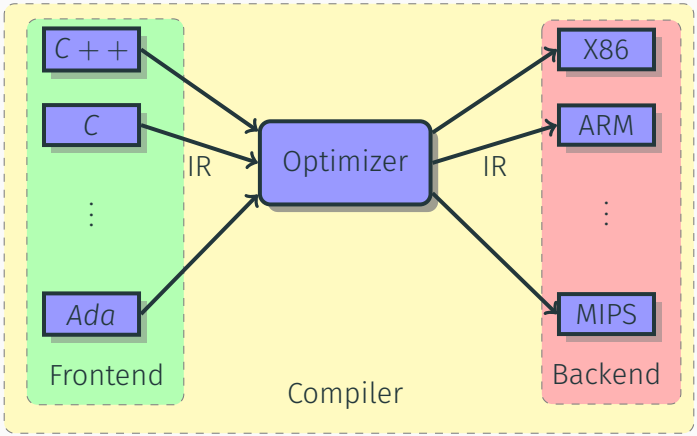
2 implementations of ICC methods in the LLVM compiler.

Propose a new **Data-Flow Approach** for Complexity Analysis.

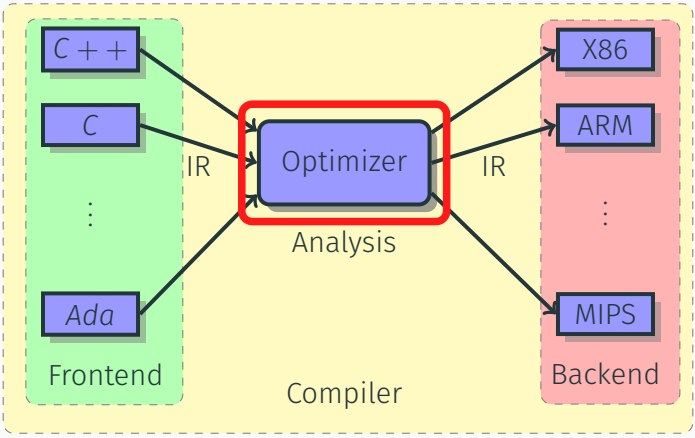
Give an instance of this Data Flow Analysis for **optimisation** :
Loop Quasi-Invariant Chunk Motion (LQICM)

Where ? In LLVM

Architecture



Architecture



LLVM Intermediate Representation

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

LLVM Intermediate Representation

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

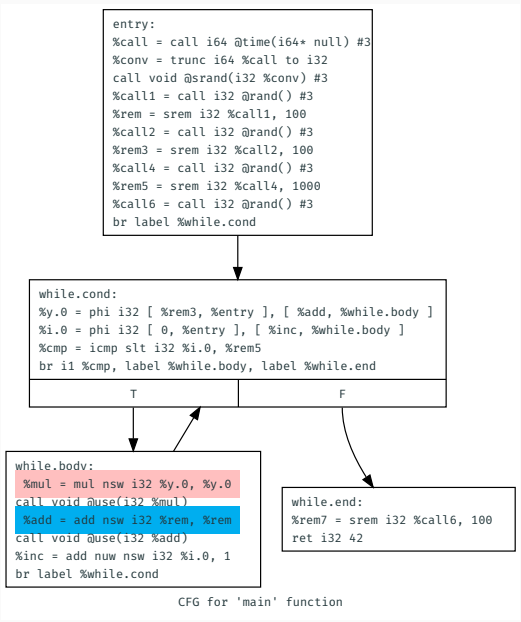
An IR is **source-language-independent**, then optimizations and analysis should work on every languages (properly translated to this IR).

LLVM Intermediate Representation

```
define i32 @main() #0 {
entry:
  %call = call i64 @time(i64* null) #3
  %conv = trunc i64 %call to i32
  call void @srand(i32 %conv) #3
  %call1 = call i32 @rand() #3
  %rem = srem i32 %call1, 100
  %call2 = call i32 @rand() #3
  br label %while.cond

while.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %while.body ]
  %y.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %exitcond = icmp ne i32 %i.0, 100
  br i1 %exitcond, label %while.body, label %while.end

while.body:
  %mul = mul nsw i32 %y.0, %y.0
  call void @use(i32 %mul)
  %add = add nsw i32 %rem, %rem
  call void @use(i32 %add)
  %inc = add nuw nsw i32 %i.0, 1
  br label %while.cond

while.end:
  %rem3 = srem i32 %call2, 100
  ret i32 42
}
```

```
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   while(i<100){
      z=y*y ;
      use(z);
      y=x+x ;
      use(y);
      i++;
   }
   return 42;
}
```

LLVM Intermediate Representation



CFG for 'main' function

```c
int main(){
   int i=0,y=0;
   srand(time(NULL));
   int x=rand()%100;
   int x2=rand()%100;
   int z;
   while(i<100){
       z=y*y ;
       use(z);
       y=x+x ;
       use(y);
       i++;
   }
   return 42;
}
```

First contribution :

Detection of Non Size Increasing Programs

Bounding Complexity

Safe recursion from S. Bellantoni and S. Cook (1992)

Bounding Complexity

Safe recursion from S. Bellantoni and S. Cook (1992)

Non Size Increasing by M. Hofmann (1999)

Bounding Complexity

Safe recursion from S. Bellantoni and S. Cook (1992)

Non Size Increasing by M. Hofmann (1999)

Let's detect programs which compute within a **constant amount of space** !

Non-Size-Increasing

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

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.

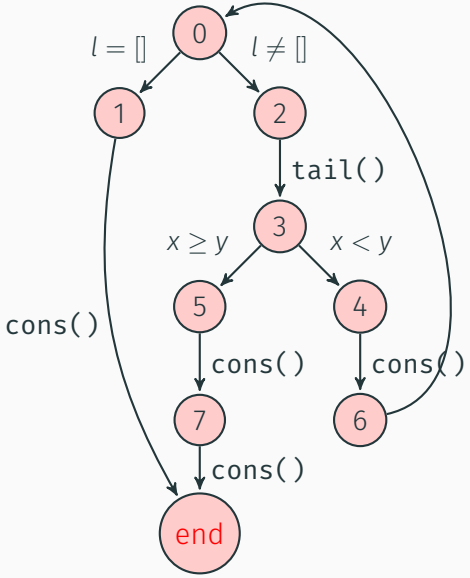
Non-Size-Increasing

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

Hofmann detects non size increasing programs by adding a special type ◊ which can be seen as the type of **pointers to free memory** in Imperative Programs.

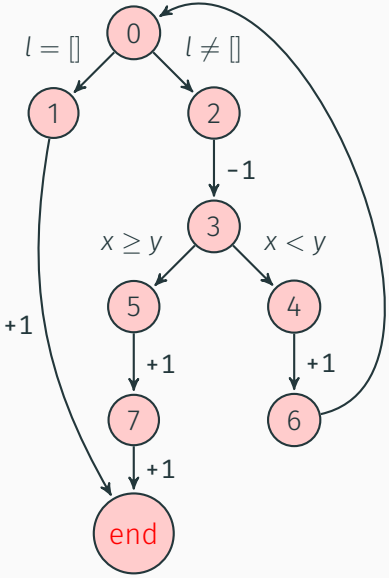
Control Flow Graph

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



Space-Resource Control Graph

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



Building Space-Resource Control Graph

LLVM tools already provide the **Control Flow Graph**

Building Space-Resource Control Graph

LLVM tools already provide the **Control Flow Graph**

Compute the weight of each **Basic Block**
It gives a **Resource Control Graph**

Building Space-Resource Control Graph

LLVM tools already provide the **Control Flow Graph**

Compute the weight of each **Basic Block**
It gives a **Resource Control Graph**

Find the heaviest path and detect **positive loops**
(with the Bellman-Ford's Algorithm)

Conclusion

2 passes :

- builds a reusable **Resource Control Graph**
- detects **positive loops**

1. https ://github.com/ThomasRuby/NSIDetectionPass

Conclusion

2 passes :

- builds a reusable **Resource Control Graph**
- detects **positive loops**

Provides an answer to the question “Is the program/function NSI ?”

1. https ://github.com/ThomasRuby/NSIDetectionPass

Conclusion

2 passes :

- builds a reusable **Resource Control Graph**
- detects **positive loops**

Provides an answer to the question “Is the program/function NSI ?”

only 200 lines of code thanks to the modularity of the compiler

1. https ://github.com/ThomasRuby/NSIDetectionPass

Conclusion

2 passes :

- builds a reusable **Resource Control Graph**
- detects **positive loops**

Provides an answer to the question “Is the program/function NSI ?”

only 200 lines of code thanks to the modularity of the compiler

Available on **github** [1]

1. https ://github.com/ThomasRuby/NSIDetectionPass

Conclusion

2 passes :

- builds a reusable **Resource Control Graph**
- detects **positive loops**

Provides an answer to the question “Is the program/function NSI ?”

only 200 lines of code thanks to the modularity of the compiler

Available on **github** [1]

A good **first implementing experience in LLVM**...

1. https ://github.com/ThomasRuby/NSIDetectionPass

A Data-Flow approach

History

From Lars Kristiansen and Neil D. Jones' work on *mwp*-Bounds.

- Different traceable **flows** more or less *"harmful"*
- "*m*" for **max**, "*w*" for **weak-polynomial** and "*p*" for **polynomial**

History

From Lars Kristiansen and Neil D. Jones' work on *mwp*-Bounds.

- Different traceable **flows** more or less *"harmful"*
- "*m*" for **max**, "*w*" for **weak-polynomial** and "*p*" for **polynomial**

Lars and Neil developed a method (using syntactic proof calculus) to compute a ***mwp*-graph** over a simple LOOP-language

History

From Lars Kristiansen and Neil D. Jones' work on *mwp*-Bounds.

- Different traceable **flows** more or less *"harmful"*
- "*m*" for **max**, "*w*" for **weak-polynomial** and "*p*" for **polynomial**

Lars and Neil developed a method (using syntactic proof calculus) to compute a ***mwp*-graph** over a simple LOOP-language

Question : **how powerful** can it be on real programs ?

A Data-Flow Analysis Framework

A **customisable** Data-Flow Analysis designed to build **Data-Flow graphs**

A **semi-ring** as input.

A Data-Flow Analysis Framework

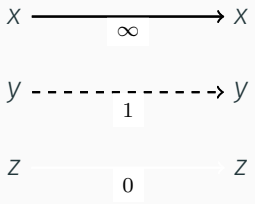
A **customisable** Data-Flow Analysis designed to build **Data-Flow graphs**

A **semi-ring** as input.

I present this framework with an example

Definition

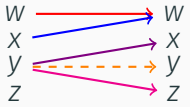
A **Data Flow Graph** represents **“influences”** between variables at different states :



for the example : the semi-ring is $(\{0, 1, \infty\}, \max, \times)$.

Matrix algebra

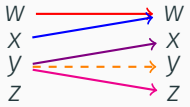
A *Data Flow Graph* for a **command** is a $n \times n$ matrix over a semi-ring



$$\rightarrow \quad \begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ 0 & \infty & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrix algebra

A *Data Flow Graph* for a **command** is a $n \times n$ matrix over a semi-ring



$$\rightarrow \quad \begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ 0 & \infty & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

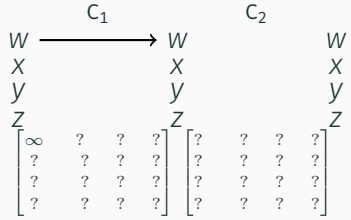
What is a **command**?

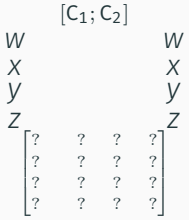
A `WHILE`-language

(Variables)	X	::=	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	*exp*	::=	$X \mid$ `op(exp,…,exp)`
(Command)	*com*	::=	X=*exp* | *com*;*com* | `skip` |
			`while` *exp* `do` *com* |
			`if` *exp* `then` *com* `else` *com* |
			`use(`X_1`,…,`X_n`)`

Multipath and Composition (*à la "size-change Termination"*)

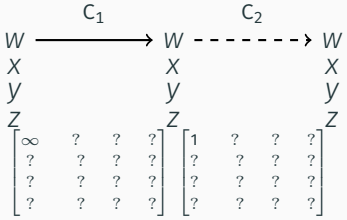
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.

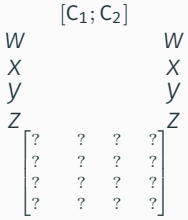




Multipath and Composition (*à la "size-change Termination"*)

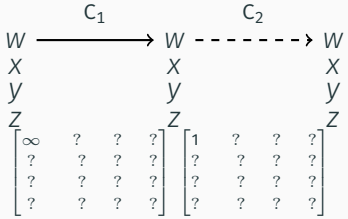
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.

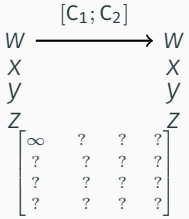




Multipath and Composition (*à la "size-change Termination"*)

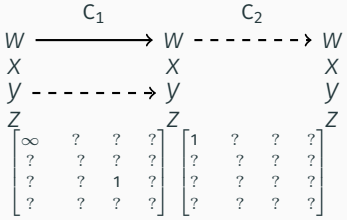
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.

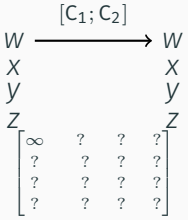




Multipath and Composition (*à la "size-change Termination"*)

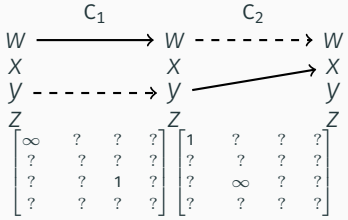
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.

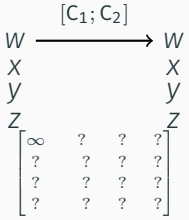




Multipath and Composition (*à la "size-change Termination"*)

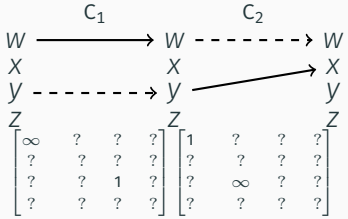
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.

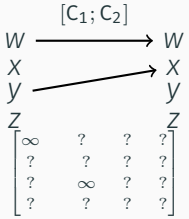




Multipath and Composition (*à la "size-change Termination"*)

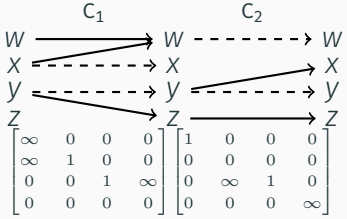
Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.



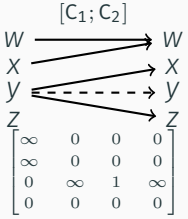


Multipath and Composition (*à la "size-change Termination"*)

Let $\mathsf{C} := [\mathsf{C}_1; \mathsf{C}_2; \ldots; \mathsf{C}_n]$;
Then $\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1)\mathrm{M}(\mathsf{C}_2)\ldots\mathrm{M}(\mathsf{C}_n)$.



$$\begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ 0 & 0 & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \infty & 1 & 0 \\ 0 & 0 & 0 & \infty \end{bmatrix}$$

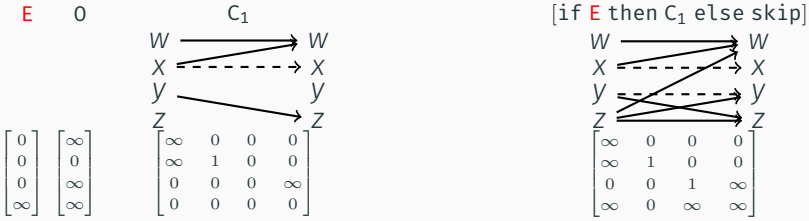


$$\begin{bmatrix} \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ 0 & \infty & 1 & \infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Condition

Let C := `if` E `then` C_1 `else skip`;

Then $\mathrm{M}(C) = \mathrm{M}(C_1) + \mathrm{Id} + (E^t O) = (\mathrm{M}(C_1) + \mathrm{Id})^{[E]}$



Compute the **max DFG** regarding to the different possible paths.

Loop `while` (à la *“mwp-polynomials”*)

Let $\mathsf{C} := \texttt{while}\ \mathsf{E}\ \texttt{do}\ \mathsf{C}_1;$

Loop `while` (à la *“mwp-polynomials”*)

Let $\mathtt{C} := \mathtt{while\ E\ do\ C_1}$;

Number of iteration unknown :

$\mathtt{skip}$ $\quad$ $\mathtt{C_1}$ $\quad$ $\mathtt{C_1; C_1}$ $\quad$ $\mathtt{C_1; C_1; C_1}$ $\quad$ etc...

Loop `while` (à la *"mwp-polynomials"*)

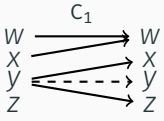
Let $\mathtt{C} := \mathtt{while\ E\ do\ C_1};$

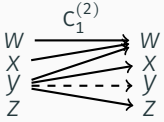
Number of iteration unknown :

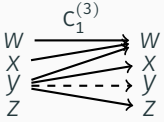
$\mathtt{skip}$ $\quad \mathtt{C_1} \quad \mathtt{C_1};\mathtt{C_1} \quad \mathtt{C_1};\mathtt{C_1};\mathtt{C_1} \quad$ etc...

Compute the max :

$$\mathrm{M}(\mathtt{C}_1^*)$$







Trivially converges by motonicity

Loop `while` (à la *“mwp-polynomials”*)

Let $\mathsf{C} := \texttt{while}\ \mathsf{E}\ \texttt{do}\ \mathsf{C}_1$;
$\mathrm{M}(\mathsf{C}) = \mathrm{M}(\mathsf{C}_1^*)^{[\mathsf{E}]}$.

What could it be used for ?

Potential Instances

- **Complexity analysis** à la *mwp*-bounds or NSI
- **Energy consumption** analysis
- … ?

Potential Instances

- **Complexity analysis** à la *mwp*-bounds or NSI
- **Energy consumption** analysis
- … ?
- Compute the **“worst” dependency graph** for hoisting

Potential Instances

- **Complexity analysis** à la *mwp*-bounds or NSI
- **Energy consumption** analysis
- … ?
- Compute the **“worst” dependency graph** for hoisting

That’s what I present now!

Instance : Loop Quasi-Invariant Chunk Motion

Loop-Invariant

```
int x=rand()%100;
while(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

Loop-Invariant

```
int x=rand()%100;
while(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

```
int x=rand()%100;
if(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
while(i<100){
   use(y);
   i=i+1;
}
```

Loop-Invariant

```
int x=rand()%100;
while(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
```

```
int x=rand()%100;
if(i<100){
   y=x+x; //invariant
   use(y);
   i=i+1;
}
while(i<100){
   use(y);
   i=i+1;
}
```

Obviously already in compilers : called "*Loop Invariant Code Motion*"
(**8062** instructions hoisted over **3808** loops in *vim*...)

Loop-Quasi-Invariants

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

Loop-Quasi-Invariants

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

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

Loop-Quasi-Invariants

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

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

Done but sequentially...

...regarding to invariants (LICM + instcombine)

Loop-Invariant Chunk

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

Loop-Invariant Chunk

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Loop-Invariant Chunk

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Definitely new! (at least in GCC and LLVM)

Loop-Quasi-Invariant Chunk

```
while(j<n){
    fact=1;        //1
    i=1;           //1
    while (i<=y) { //3
        fact=fact*i;
        i=i+1;
    }
    if(x<100)      //2
        y=x+a;
    if(x>=100)     //1
        y=x+100;
    a = 12;        //1
    i=j+1;         //∞
    j=i+1;         //∞
}
```

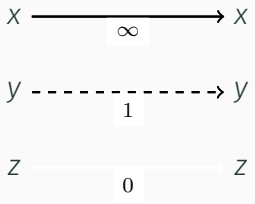
Loop-Quasi-Invariant Chunk

```
while(j<n){
    fact=1;          //1
    i=1;             //1
    while (i<=y) {   //3
        fact=fact*i;
        i=i+1;
    }
    if(x<100)        //2
        y=x+a;
    if(x>=100)       //1
        y=x+100;
    a = 12;          //1
    i=j+1;           //∞
    j=i+1;           //∞
}
```

```
if (j < n){
    fact = 1;
    â0 = fact;
    i = 1;
    â2 = i;
    while (i <= y){
        fact = fact * i;
        i = i + 1;
    }
    if (x < 100)
        y = x + a;
    if (x >= 100)
        y = x + 100;
    â4 = y;
    â5 = x >= 100;
    a = 12;
    i = j + 1;
    j = i + 1;
}
if (j < n){
    fact = â0;
    i = â2;
    while (i <= y){
        fact = fact * i;
        i = i + 1;
    }
    if (x < 100)
        y = x + a;
    if (â5)
        y = â4;
    i = j + 1;
    j = i + 1;
}
if (j < n){
    fact = â0;
    i = â2;
    while (i <= y){
        fact = fact * i;
        i = i + 1;
    }
    i = j + 1;
    j = i + 1;
}
while (j < n){
    i = j + 1;
    j = i + 1;
}
```

Dependency Flow Graph (DepFG)

A **Data Flow Graph** represents **“influences”** between variables at different states :



for the example : the semi-ring is $(\{0, 1, \infty\}, \max, \times)$.

Dependency Flow Graph (DepFG)

A **Dependency Flow Graph** represents **dependencies** between vars in diff. states :

$C := [x = x + 1;$ — $x \xrightarrow[\infty]{\text{dependence}} x$

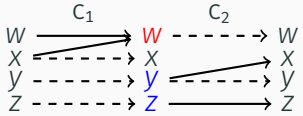
$y = y;$ — $y \xrightarrow[1]{\text{propagation}} y$

$z = 0;]$ — $z \xrightarrow[0]{\text{reinitialization}} z$

for the next : the semi-ring is $(\{0, 1, \infty\}, \max, \times)$.

Independence of chunks

Definition (independence)
If $\mathrm{Out}(C_1) \cap \mathrm{In}(C_2) = \emptyset$ *then* C_2 *is* ***independent*** *from* C_1. *This is denoted* $C_1 \prec C_2$.



Notion of Invariance degree

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

```
if(i<100){ //if¹
  z=y*y;
  use(z);
  y=x+x;
  use(y);
  i=i+1;
}
if(i<100){ //if²
  z=y*y;
  use(z);
  use(y);
  i=i+1;
}
while(i<100){ //while∞
  use(z);
  use(y);
  i=i+1;
}
```

Notion of Invariance degree

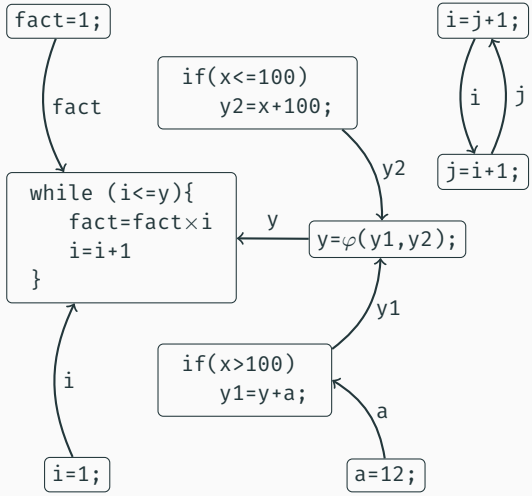
```
while(i<100){
    z=y*y; // 2
    use(z); //∞
    y=x+x; // 1
    use(y); //∞
    i=i+1; //∞
}
```

```
if(i<100){ //if¹
  z=y*y;
  use(z);
  y=x+x;
  use(y);
  i=i+1;
}
if(i<100){ //if²
  z=y*y;
  use(z);
  use(y);
  i=i+1;
}
while(i<100){ //while^∞
  use(z);
  use(y);
  i=i+1;
}
```

$$\texttt{if}^i = \texttt{if E then } [\check{C_1}; \check{C_2}; \ldots; \check{C_n}]^{(i)}, \text{ and}$$
$$\texttt{while}^i = \texttt{while E then } [\check{C_1}; \check{C_2}; \ldots; \check{C_n}]^{(i)}$$

```
while(j<n){
   fact=1;       //1
   i=1;          //1
   while (i<=y) { //3
      fact=fact*i;
      i=i+1;
   }
   if(x<100)     //2
      y1=x+a;
   if(x>=100)    //1
      y2=x+100;
   y=φ(y1,y2) //2
   a = 12;       //1
   i=j+1;        //∞
   j=i+1;        //∞
}
```

```
while(j<n){
   fact=1;        //1
   i=1;           //1
   while (i<=y) { //3
      fact=fact*i;
      i=i+1;
   }
   if(x<100)      //2
      y1=x+a;
   if(x>=100)     //1
      y2=x+100;
   y=φ(y1,y2) //2
   a = 12;        //1
   i=j+1;         //∞
   j=i+1;         //∞
}
```



Generalisation

Theorem

$$\textit{Let } \mathsf{C} := \texttt{while } \mathsf{E} \texttt{ do } [\mathsf{C}_1; \mathsf{C}_2; \ldots, \mathsf{C}_n] :$$

$$[\![\mathsf{C}]\!] \equiv [\![\texttt{if}^1; \texttt{if}^2; \ldots; \texttt{if}^{\mathrm{maxdeg}(\mathsf{C})}; \texttt{while}^\infty]\!]$$

Implementation

A prototype on LLVM tool chain

We implemented a pass in LLVM :

- Currently around 3000 lines of C++ thanks to LLVM's modularity.
- tested on several targeted examples.
- generates statistics while compiling.

Available on github [2]

2. https ://github.com/ThomasRuby/LQICM_pass

Analysis : Degree on each Instruction

```
...
while.body:
  %mul = mul nsw i32 %y.0, %y.0   ← 2
  call void @use(i32 %mul)        ← ∞
  %add = add nsw i32 %rem, %rem   ← 1
  call void @use(i32 %add)        ← ∞
  %inc = add nuw nsw i32 %i.0, 1  ← ∞
  br label %while.cond            ← ∞
...
```

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

Analysis : Degree on each Instruction

```
...
while.body:
  %mul = mul nsw i32 %y.0, %y.0   ← 2
  call void @use(i32 %mul)        ← ∞
  %add = add nsw i32 %rem, %rem   ← 1
  call void @use(i32 %add)        ← ∞
  %inc = add nuw nsw i32 %i.0, 1  ← ∞
  br label %while.cond            ← ∞
...
```

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

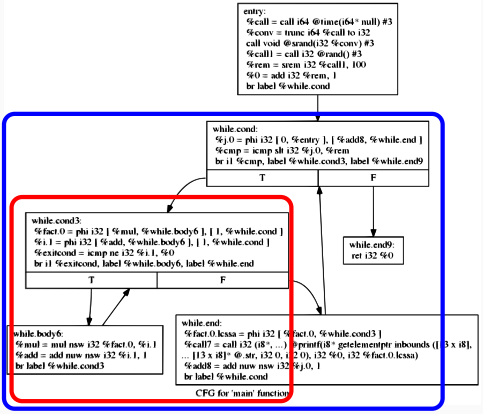
Consider all **`call`** as anchors

Factorial example peeled

```
while(j<100){
    fact=1;
    i=1;
    while(i<=n){
        fact=fact*i;
        i=i+1;
    }
    use(fact);
    j=j+1;
}
```

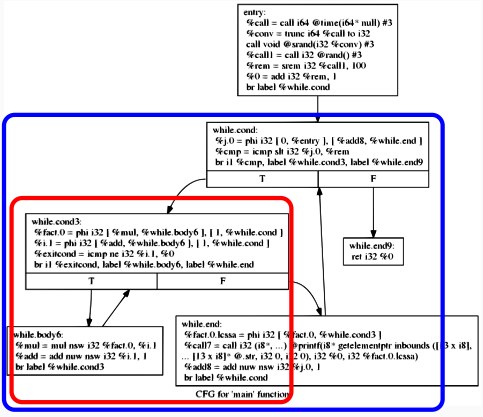
```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

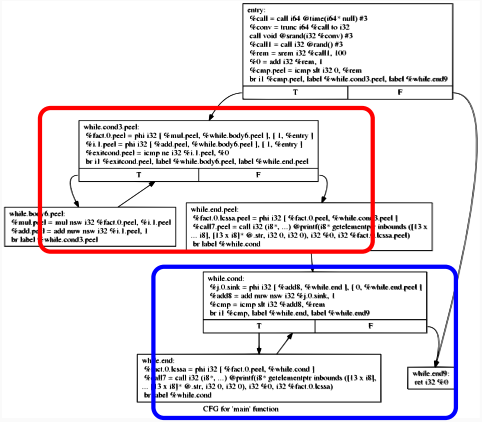
Factorial example peeled




```
if(j<100){
  fact=1;
  i=1;
  while(i<=n){
    fact=fact*i;
    i=i+1;
  }
  use(fact);
  j=j+1;
}
while(j<100){
  use(fact);
  j=j+1;
}
```

Factorial example peeled





LLVM statistics

```
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

 2 globalopt        - Number of globals deleted
 5 globalopt        - Number of globals marked unnamed_addr
 1 indvars          - Number of congruent IVs eliminated
 2 indvars          - Number of loop exit tests replaced
 2 indvars          - Number of exit values replaced
10 instcombine      - Number of insts combined
 1 instsimplify     - Number of redundant instructions removed
 8 lcssa            - Number of live out of a loop variables
 1 licm             - Number of instructions hoisted out of loop
 2 loop-rotate      - Number of loops rotated
11 loop-simplify    - Number of pre-header or exit blocks inserted
 2 loop-unswitch    - Total number of instructions analyzed
 1 loop-vectorize   - Number of loops analyzed for vectorization
 2 lqicm            - Number of time runOnLoop is performed…
 3 lqicm            - Number of instructions with deg != infty
 1 lqicm            - Number of innerBlocks with deg != infty
 4 mem2reg          - Number of PHI nodes inserted
 2 mem2reg          - Number of alloca's promoted with a single store
 9 simplifycfg      - Number of blocks simplified
```

LLVM statistics

(LQICM Analysis called before each LICM(3X) occurence)
Compiler: clang release_40 -Oz
Time (with - without)

8m8,020s - 7m48,244s	
---	---
3808	number of loops at `EP_ModuleOptimizerEarly`
13178	total number of times `lqicm LoopPass` was launched
8062	number of all *Instructions* hoisted by `licm`
160	number of instructions sunk by `licm`
6525	total number of times `lqicm` has analyzed the loop (over 13178)
7632	number of *all* quasi-invariants detected
808	number of quasi-invariants *Chunk* detected
6824	number of quasi-invariants *Instructions*
43	number of chunks with invariance degree >= 2

$\sim 50\%$ loops analysed.
Already $\sim 85\%$ of the total invariants detected by LICM.
but 808 quasi-invariants **chunks** found!

Conclusion and further work

We implemented an instance of a tunable data-flow analysis for **improving loop-invariant detection**

TODO list :

- Improve **analysis flexibility**.
- Improve **analysis time** (refactoring needed).
- Make the **transformation more stable**.
- Start benchmarking with the transformation and **compete with LICM**!

Final Conclusion

Final Conclusion

A **pillar for the bridge between two communities** : a customizable tool for data-flow analysis.

One step for **“Complexity-certifying Compiler”**.

Welcome to **“real” world** programs (then more **pessimistic**).

A promising **LLVM contribution** : improving Loop Invariant Code Motion

Publications

2016 — "Detection of Non-Size Increasing Programs in Compilers"
Jean-Yves Moyen & Thomas Rubiano
7th International Workshop on Developments in Implicit Computational Complexity (DICE 2016), Eindhoven

Apr 2017 — "Loop Quasi-Invariant Chunk Motion by peeling with statement composition"

Jean-Yves Moyen, Thomas Rubiano & Thomas Seiller
In Guillaume Bonfante and Georg Moser : Proceedings 8th Workshop on *Developments in Implicit Computational complExity* and 5th Workshop on *FOundational and Practical Aspects of Resource Analysis* (DICE-FOPARA 2017), Uppsala, Sweden, April 22-23, 2017, Electronic Proceedings in Theoretical Computer Science 248, pp. 47–59.

Oct 2017 — "Loop Quasi-Invariant Chunk Detection"

Jean-Yves Moyen, Thomas Rubiano & Thomas Seiller
15^{th} *International Symposium on Automated Technology for Verification and Analysis* (ATVA 2017)
Pune

Implicit Computational Complexity meets Compilers

funded by ELICA (ANR-14-CE25-0005)

Thomas Rubiano

Jury :

Guillaume Bonfante	Université de Lorraine	Rapporteur
Christophe Fouqueré	Université Paris13	Examinateur
Laure Gonnord	Université Lyon1	Rapporteuse
Tobias Grosser	ETH Zürich	Examinateur
Stefano Guerrini	Université Paris13	Examinateur
Virgile Mogbil	Université Paris13	Directeur
Torben Mogensen	Københavns Universitet	Examinateur
Jean-Yves Moyen	Université Paris13	Encadrant
Ulrich Schöpp	Ludwig-Maximilians-Universität München	Examinateur
Jakob Grue Simonsen	Københavns Universitet	Directeur

Definition
Let $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \ldots, \mathtt{C_n}]$. *We define the directed graph* $\mathrm{Dep}(\mathtt{C})$ *as follows :*

- *the set of vertices* $V^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\{\mathtt{C_1}, \ldots, \mathtt{C_n}\}$
- *the set of edges* $E^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\uplus_{m=1}^{n} \uplus_{i \in \mathrm{In}(\mathtt{C_m})} \mathrm{PrD}_i(\mathtt{C_m})$
- *the source* $s(i)$ *of the edge* $\mathtt{C_k} \in \mathrm{PrD}_i(\mathtt{C_m})$ *is* $\mathtt{C_k}$*;*
- *the target* $t(i)$ *of the edge* $\mathtt{C_k} \in \mathrm{PrD}_i(\mathtt{C_m})$ *is* $\mathtt{C_m}$.

Moving Independent Chunks

Definition (Mutual Independence)
If $\mathsf{C}_2 \prec \mathsf{C}_1$ *and* $\mathsf{C}_1 \prec \mathsf{C}_2$*, we say that* C_2 *and* C_1 *are* ***mutually independents****, and write* $\mathsf{C}_1 \asymp \mathsf{C}_2$*.*

Lemma (Swapping commands)
If $\mathsf{C}_1 \asymp \mathsf{C}_2$*, then :*

$$[\![\mathsf{C}_1; \mathsf{C}_2]\!] \equiv [\![\mathsf{C}_2; \mathsf{C}_1]\!]$$

Lemma (Hoisting mutual independent commands)
If C_1 *is self-independent (i.e.* $\mathsf{C}_1 \asymp \mathsf{C}_1$*),* $\mathrm{Var}(\mathsf{E}) \cap \mathrm{Out}(\mathsf{C}_1) = \emptyset$*, and if* $\mathsf{C}_1 \asymp \mathsf{C}_2$*, then :*

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

Moving Independent Chunks

Definition (Mutual Independence)
If $C_2 \prec C_1$ *and* $C_1 \prec C_2$, *we say that* C_2 *and* C_1 *are* ***mutually independents****, and write* $C_1 \asymp C_2$.

Lemma (Swapping commands)
If $C_1 \asymp C_2$, *then :*

$$[\![C_1; C_2]\!] \equiv [\![C_2; C_1]\!]$$

Lemma (Hoisting mutual independent commands)
If C_1 *is self-independent (i.e.* $C_1 \asymp C_1$*),* $\mathrm{Var}(E) \cap \mathrm{Out}(C_1) = \emptyset$, *and if* $C_1 \prec C_2$, *then :*

$$[\![\texttt{while } E \texttt{ do } [C_1; C_2]]\!] \equiv [\![\texttt{if } E \texttt{ then } [C_1; C_2]; \texttt{while } E \texttt{ do } [C_2]]\!]$$

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
   4.1 **IF** `cd <= dd` and the current command dominates this dependence **THEN** `cd = dd + 1`
   ELSE `cd = dd`

```
---- MapDegOfOC ----
%i.0 = phi i32 [ undef, %entry ], [ %j.0, %while.end ] = 2
%j.0 = phi i32 [ 0, %entry ], [ %add20, %while.end ] = -1
%y.0 = phi i32 [ 5, %entry ], [ %y.2, %while.end ] = 3
%a.0 = phi i32 [ 5, %entry ], [ 0, %while.end ] = 2
br label %while.cond5 = -1
%fact.0 = phi i32 [ %mul, %while.body8 ], [ 1, %while.cond5.preheader ] = 1
%i.1 = phi i32 [ %add, %while.body8 ], [ 1, %while.cond5.preheader ] = 1

while.cond5:                                      ; preds = %while.cond5.preheader, %while.body8
  %fact.0 = phi i32 [ %mul, %while.body8 ], [ 1, %while.cond5.preheader ]
  %i.1 = phi i32 [ %add, %while.body8 ], [ 1, %while.cond5.preheader ]
  %cmp6 = icmp sle i32 %i.1, %y.0
  br i1 %cmp6, label %while.body8, label %while.end
 = 4

It's the beginning of an inner Chunk ↑

%fact.0.lcssa = phi i32 [ %fact.0, %while.cond5 ] = 4
%i.1.lcssa = phi i32 [ %i.1, %while.cond5 ] = 4
%call9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str, i32 0, i32 0), i32 %i.1.l
%cmp10 = icmp sgt i32 %rem3, 100 = 1
%add12 = add nsw i32 %rem3, %a.0 = 3
%add12.y.0 = select i1 %cmp10, i32 %add12, i32 %y.0 = 3
%call13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str.1, i32 0, i32 0), i32 %ad
%cmp14 = icmp sle i32 %rem3, 100 = 1
%add17 = add nsw i32 %rem3, 100 = 1
%y.2 = select i1 %cmp14, i32 %add17, i32 %add12.y.0 = 3
%call19 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str.1, i32 0, i32 0), i32 %y.
%add20 = add nuw nsw i32 %j.0, 1 = -1
br label %while.cond = -1
--------------------
```