Loop Quasi-Invariant Chunk Motion by peeling with statement composition

Jean-Yves Moyen[1], Thomas Rubiano[1,2], and Thomas Seiller[1]

[1] Department of Computer Science University of Copenhagen (DIKU)
[2] Université Paris 13 - LIPN

Abstract. Several techniques for analysis and transformations are used in compilers. Among them, the peeling of loops for hoisting quasi-invariants can be used to optimize generated code, or simply ease developers' lives. In this paper, we introduce a new concept of dependency analysis borrowed from the field of Implicit Computational Complexity (ICC), allowing to work with composed statements called "Chunks" to detect more quasi-invariants. Based on an optimization idea given on a `WHILE` language, we provide a transformation method - reusing ICC concepts and techniques [8,10] - to compilers. This new analysis computes an invariance degree for each statement or chunks of statements by building a new kind of dependency graph, finds the "maximum" or "worst" dependency graph for loops, and recognizes if an entire block is Quasi-Invariant or not. This block could be an inner loop, and in that case the computational complexity of the overall program can be decreased. We already implemented a proof of concept on a toy C parser[3] analysing and transforming the AST representation.
In this paper, we introduce the theory around this concept and present a prototype analysis pass implemented on LLVM. In a very near future, we will implement the corresponding transformation and provide benchmarks comparisons.

Keywords: Static Analysis, Transformations, Optimization, Compilers, Loop Invariants, Complexity, Quasi-Invariants

1 Introduction

A compiler turns some high-level program into a (semantically) equivalent low-level assembly program. This translation implies many smaller transformations, notably because features such as objects, exceptions, or even loops need to be expressed in assembly language. The compiler also performs many optimisations aiming at making the code more efficient. These are often needed to streamline the code generated by the transformations but can also be used to optimise the source code.

A command inside a loop is *loop invariant code* if its execution has no effect after the first iteration of the loop. Typically, an assignment `x:=0` in a loop is invariant (provided `x` is not modified elsewhere). Loop invariants can safely be moved out of loops (*hoisted*) in order to make the program run faster.

[3] `https://github.com/ThomasRuby/LQICM_On_C_Toy_Parser`

While loop invariant code is maybe not so frequent in source code, many transformations along the compilation process can generate some. For example, when compiling the editors `vim` or `emacs`, an average of 10 commands per loop can be hoisted. These are mostly generated by other optimisations.

A command inside a loop is *quasi-invariant* if its execution has no effect after a finite number of iterations of the loop. Typically, if a loop contains the sequence `x:=y, y:=0`, then `y:=0` is invariant. However, `x:=y` is **not** invariant. The first time the loop is executed, `x` will be assigned the old value of `y`, and only from the second time onward will `x` be assigned the value `0`. Hence, this command is *quasi-invariant*. It can still be hoisted out of the loop, but to do so requires to *peel* the loop first, that is execute its body once (by copying it before the loop).

The number of times a loop must be executed before a quasi-invariant can be hoisted is called here the *degree* of the invariant.

An obvious way to detect quasi-invariants is to first detect invariants (that is, quasi-invariants of degree 1) and hoist them; and iterate the process to find quasi-invariant of degree 2, and so on. This is, however, not very efficient since it may require a large number of iterations to find some invariance degrees.

We provide here an analysis able to directly detect the invariance degree of any statements in the loop. Moreover, our analysis is able to assign an invariance degree non only to individual statements but also to groups of statements (called *chuncks*). That way it is possible, for example, to detect that a whole inner loop is invariant and hoist it, thus decreasing the asymptotic complexity of the program.

This analysis and transformation has first been implemented as a *Proof of Concept* in a toy `C`-parser. Next, the analysis has been implemented as a prototype *pass* of the mainstream compiler LLVM and the transformation is under way. The prototype is currently unable to handle several common situations (and leave them untouched) because of choices made for the sake of simplicity. It is, nonetheless, powerful enough to make significant progress compared to the existing loop invariant code motion techniques (it can handle many more cases).

Loop optimization techniques based on quasi-invariance are well-known in the compilers community. The transformation idea is to peel loops a finite number of time and hoist invariants until there are no more quasi-invariants. As far as we know, this technique is called “peeling” and it was introduced by Song *et al.* [13].

Loop peeling and unrolling can also happen for entirely different reasons, mostly to optimise pipelines. In these cases, the decision to unroll is based on loop size and (predicted) number of iterations but not on the presence of quasi-invariants. It may, of course, happen that quasi-invariant removal is performed as a side effect of this unrolling, but only as a side effect and not as the main goal.

The present paper offers a new point of view on invariant and quasi-invariant detection. Adapting ideas from an optimization on a `WHILE` language by Lars Kristiansen [8], we provide a way to compute invariance degrees based on techniques developed in the field of Implicit Computational Complexity.

Implicit Computational Complexity (ICC) studies computational complexity in terms of restrictions of languages and computational principles, providing results that do not depend on specific machine models. Based on static analysis, it helps predict and control

resources consumed by programs, and can offer reusable and tunable ideas and techniques for compilers. ICC mainly focuses on syntactic [4,3], type [6,2] and Data Flow [11,7,9,12] restrictions to provide bounds on programs' complexity. The present work was mainly inspired by the way ICC community uses different concepts to perform Data Flow Analysis, e.g. "Size-change Graphs" [11] or "Resource Control Graphs"[12] which track data values' behavior and use a matrix notation inspired by [1], or "mwp-polynomials" [9] to provide bounds on data size.

For our analysis, we focus on dependencies between variables to detect invariance. Dependency graphs [10] can have different types of arcs representing different kind of dependencies. Here we will use a kind of Dependence Graph Abstraction [5] that can be used to find local and global quasi-invariants. Based on these techniques, we developed an analysis pass and we will implement the corresponding transformation in LLVM.

We propose a tool which is notably able to give enough information to easily peel and hoist an inner loop, thus decreasing the complexity of a program from $O(n^2)$ to $O(n)$.

1.1 State of the art on Quasi-Invariant detection in loop

Modern compilers find loop invariant code by recursively searching for variables whose value only depends on either code that is outside the loop; or other loop invariant code. To our knowledge, no compiler searches for loop quasi-invariant code.

A quasi-invariance detection has been described in [13]. The authors define a *variable dependency graph* (VDG) and detect a loop quasi-invariant variable `x` if, among all paths ending at `x`, no path contain a node included in a circular path. Then they deduce an *invariant length* which corresponds to the length of the longest path ending in `x`. To our knowledge, this analysis has not been implemented in a compiler. Moreover, they only analyse individual commands and do not handle chunks. In the present paper, this *length* is called *invariance degree*.

1.2 Contributions

This paper lies between the fields of Implicit Computational Complexity and Compilation and provides significant advancement to both.

To the authors' knowledge, this is the first application of ICC techniques on a mainstream compiler. One interest is that our tool potentially applies to programs written in any programming language managed by LLVM. Moreover, this work should be considered as a first step of a larger project that will make ICC techniques more accessible to programmers. Thus, we show that 25 years after Bellantoni & Cook breakthrough [3], ICC techniques are ready to move into "the real world".

On a more technical side, our tool aims at improving on currently implemented loop invariant detection and optimization techniques. The main LLVM tool for this purpose, the Loop Invariant Code Motion pass (LICM), does not detect quasi-invariant of degree more than 3 (and not all of those of degree 2). More importantly, LICM will not detect quasi-invariant blocks of code (that we call *chunk*), such as whole loops. Our tool, on the other hand, detects quasi-invariants of arbitrary degree and is able to deal with chunks.

For instance the optimization shown in Figure 5 is performed neither by LLVM nor by GCC even at their maximum optimization level.

2 Data Flow Graphs

In this section, we sketch the main lines of the theory of *data flow graphs*. While in later sections we will only be studying a specific case of those, the theory is quite general and pinpoints to formal links with various works on static analysis [11,1,9], and programming languages semantics [**?**,**?**].

Here data flow graphs are used to represent (weighted) relations between variables, that is relations that carry some additional information represented by elements of a *semi-ring*. In the next section, for instance, the semi-ring[4] $(\{0, 1, \infty\}, \max, \times)$ will be used to represent various kinds of dependencies between variables. Consequently, all examples will be given with this specific choice of semi-ring.

2.1 Definition of Data Flow Graphs

We will work with a simple imperative `WHILE`-language, with semantics similar to `C`. The grammar is given by:

(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* $\mid$ *com*`;`*com* $\mid$ `skip` $\mid$ `while` *exp* `do` *com* `od` $\mid$
`if` *exp* `then` *com* `else` *com* `fi` $\mid$ `use(`X_1`,...,`X_n`)`

A `WHILE` program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call* or a *skip*. The *use* command represents any command which does not modify its variables but use them and should not be moved around carelessly (typically, a `printf`). In practice, we currently treat all function calls as *use*, even if the function is pure. *Statements* are abstracted into *commands*. A *command* can be a statement or a sequence of commands. We also call a sequence of commands a *chunk*.

A data-flow graph for a given command `C` will be a weighted relation on the set V of variables involved in `C`. Formally, this can be represented as a matrix over a semi-ring, with the implicit choice of a denumeration of the set V. We now fix, until the end of this section, an arbitrary semi-ring $(\mathcal{S}, +, \times)$.

Definition 1 *A* Data Flow Graph *(*DFG*) for a command* `C` *is a* $n \times n$ *matrix over the semi-ring* $(\mathcal{S}, +, \times)$ *where* n *is the number of variables involved in* `C`*.*

We write $\mathrm{M}(\mathtt{C})$ *the* DFG *of* `C`*.*

For technical reasons, we identify the DFG of a command `C` with any embedding of $\mathrm{M}(\mathtt{C})$ in a larger matrix. I.e. we will abusively call the DFG of `C` any matrix of the form

$$\begin{pmatrix} \mathrm{M}(\mathtt{C}) & 0 \\ 0 & \mathrm{Id} \end{pmatrix},$$

[4] The convention here is that $0 \times \infty = 0$.

implicitly viewing the additional rows/columns as variables that do not appear in C.

In all examples, we will be using weighted relations, or weighted bi-partite graphs, to illustrate these matrices. Moreover, we will always use the semi-ring $(\{0, 1, \infty\}, \max, \times)$, since it is the specific case that will be under study in later sections: it will be used to represent dependencies: 0 will represent *reinitialization*, 1 will represent *propagation*, and ∞ will represent *dependence*. Figure Figure 1 introduces both these notions and the graphical convention used throughout this paper.

Graphically, dependencies are represented by two types of arrows from variables on the left to variables on the right: plain arrows for *direct dependency*, dashed arrows for *propagation*. *Reinitialisation* of a variable z then corresponds to the absence of arrows ending on the right occurrence of z. Figure 1 illustrates these types of dependencies; let us stress here that the DFG would be the same if the assignment $y = y;$ were to be removed[5] from C since the value of y is still propagated.

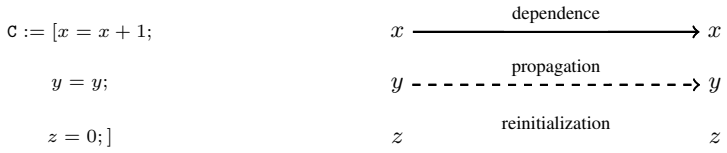


Fig. 1: Types of dependence

For convenience we define, given a command C, the following two sets of variables.

Definition 2 *Let* C *be a command. We define* $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*) as the set of variables* used *(resp.* modified*) by* C*.*

Note that in the case of dependencies, $\mathrm{In}(\mathtt{C})$ is exactly the set of variables that are source of a "dependence" arrow, while $\mathrm{Out}(\mathtt{C})$ is the set of variables that either are targets of dependence arrows or were reinitialised.

2.2 Constructing DFGs

We now describe how the DFG of a command can be computed by induction on the structure of the command. Base cases (skip, use and assignment) should be defined depending on what the DFG will be used for. When representing *dependencies*, as shown in Figure 1, we simply formalise the implicit definitions used in the figure to define the DFG in the case of assignments, define a variable e not part of the language, and define:

- $\mathrm{M}(\mathtt{skip})$ as the empty matrix[6];
- $\mathrm{M}(\mathtt{use}(\mathtt{X_1}, \ldots, \mathtt{X_n}))$ as the matrix with coefficients from each X_i and e to e equal to ∞, and 0 coefficients otherwise.

[5] Note that $y = y;$ does not create a direct dependence

[6] Up to the identification of the DFG with its embedding, it is therefore the "zero matrix" of any size, i.e. the matrix with only 0 coefficients.

Composition and Multipaths. We now turn to the definition of the DFG for a (sequential) *composition* of commands. This abstraction allows us to see a block of statements as one command with its own DFG.

Definition 3 *Let* $\mathtt{C}$ *be a sequence of commands* $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$. *Then* $\mathrm{M}(\mathtt{C})$ *is defined as the matrix product* $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.

Following the usual product of matrices, the product of two matrices A, B is defined here as the matrix C with coefficients: $C_{i,j} = \sum_{k=1}^{n}(A_{i,k} \times B_{k,j})$.

It is standard that the product of matrices of weights of two graphs F, G represents a graph of length 2 paths. This operation of matrix multiplication corresponds to the computation of *multipaths* [11] in the graph representation of DFGs. We illustrate this intuitive construction on an example in Figure 2.

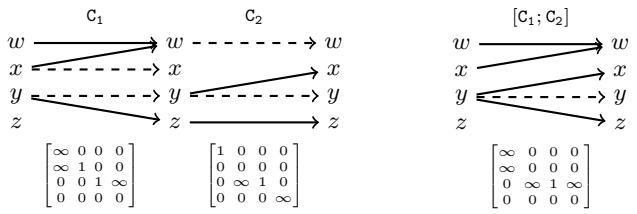


Fig. 2: DFG of Composition.

Here $\mathtt{C}_1 := [w = w + x; z = y + 2;]$ and $\mathtt{C}_2 := [x = y; z = z * 2;]$

Conditionals. We now explain how to compute the DFG of a conditional, i.e. we define the DFG of $\mathtt{C} := \mathtt{if\ E\ then\ C_1\ else\ C_2};$ from the DFG of the commands $\mathtt{C}_1$ and $\mathtt{C}_2$.

Firstly, we need to take into account that both commands $\mathtt{C}_1$ and $\mathtt{C}_2$ may be executed. In that case, the overall command $\mathtt{C}$ should be represented by the sum $\mathrm{M}(\mathtt{C}_1) + \mathrm{M}(\mathtt{C}_2)$.

However, in most cases, it is not enough to consider $\mathrm{M}(\mathtt{C}_1)+\mathrm{M}(\mathtt{C}_2)$, and the DFG of the command $\mathtt{C}$ should be obtained by adding a *conditional correction* that may depend on the expressions $\mathtt{E}$ and $\mathtt{C}$. This correction will here be written as $\mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

In the case of dependencies, we can notice that all modified variables in $\mathtt{C}_1$ and $\mathtt{C}_2$ should depend on the variables used in $\mathtt{E}$. Denoting E the vector representing variables in[7] $\mathrm{Var}(\mathtt{E})$, O the vector representing variables in $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$, and $(\cdot)^t$ the matrix transpose, we define $\mathcal{C}^{\mathtt{C}}(\mathtt{E}) = E^t O$. Figure 3 illustrates this on an example.

Definition 4 *Let* $\mathtt{C} = \mathtt{if\ E\ then\ C_1\ else\ C_2};$. *Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C}_1) \oplus \mathrm{M}(\mathtt{C}_2) \oplus \mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

[7] I.e. the vector with a coefficient equal to 1 for the variables in $\mathrm{Var}(\mathtt{E})$, and $-\infty$ for all others variables.

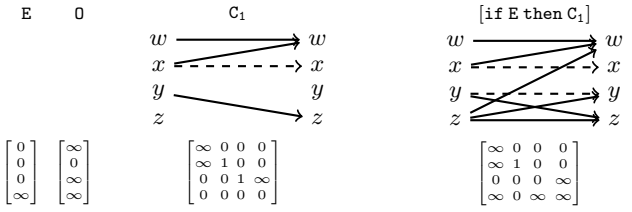


Fig. 3: DFG of Conditional.

Here $\mathtt{E} := z \geq 0$ and $\mathtt{C}_1 := [w = w + x; z = y + 2; y = 0;]$;

While Loop Finally, we define the DFG of a command $\mathtt{C}$ of the form $\mathtt{C} := \mathtt{while\ E\ do\ C_1;}$. Again, this definition splits into two steps. First, we define a matrix $\mathrm{M}(\mathtt{C}_1^*)$ representing iterations of the command $\mathtt{C}_1$; then we introduce a *loop correction* $\mathcal{W}^{\mathtt{C}}(\mathtt{E})$. In the case of dependencies, the loop correction and the conditional correction coincide: $\mathcal{W}^{\mathtt{C}}(\mathtt{E}) = \mathcal{C}^{\mathtt{C}}(\mathtt{E})$.

When considering iterations of $\mathtt{C}_1$, the first occurrence of $\mathtt{C}_1$ will influence the second one and so on. Computing the DFG of $\mathtt{C}_1^n$, the n-th iteration of $\mathtt{C}_1$, is just computing the power of the corresponding matrix, i.e. $\mathrm{M}(\mathtt{C}_1^n) = \mathrm{M}(\mathtt{C}_1)^n$. But since the number of iteration cannot be decided *a priori*, we need to sum over all possible values of n. The following expression then defines the DFG of the (informal) command $\mathtt{C}_1^*$ corresponding to "iterating $\mathtt{C}_1$ a finite (but arbitrary) number of times":

$$\mathrm{M}(\mathtt{C}_1^*) = \mathrm{limit}_{k\to\infty} \sum_{i=1}^{k} \mathrm{M}(\mathtt{C}_1)^i$$

To ease notations, we note $\mathrm{M}(\mathtt{C}_1)^{(k)}$ the partial summations $\sum_{i=1}^{k} \mathrm{M}(\mathtt{C}_1)^i$.

Definition 5 *Let* $\mathtt{C} = \mathtt{while\ E\ do\ C_1;}$. *Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C}_1^*) + \mathcal{W}^{\mathtt{C}}(\mathtt{E})$.

Since the set of all relations is finite and the sequence $(\mathrm{M}(\mathtt{C}_1)^{(k)})_{k\geqslant 0}$ is monotonic, this sequence is eventually constant. I.e., there exists a natural number N such that $\mathrm{M}(\mathtt{C}_1)^{(k)} = \mathrm{M}(\mathtt{C}_1)^{(N)}$ for all $k \geqslant N$. In the case of the semiring $(\{0, 1, \infty\}, \max, \times)$, one can even obtain a reasonable bound on the value of N.

Lemma 1 *Let* $\mathtt{C}$ *be a command, and* $K = \min(i, o)$*, where* i *(resp.* o*) denotes the number of variables in* $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*). Then, the sequence* $(\mathrm{M}(\mathtt{C}^{(k)}))_{k\geqslant K}$ *is constant.*

3 Dependencies and Quasi-Invariants

We now study in more details the DFG representation of programs for the semiring $(\{0, 1, \infty\}, \max, \times)$. Each different weight – or type of arrow – represents different types of dependencies.

Each weight express how the values of the involved variables *after* the execution of the command depend on their values *before* the execution. There is a *direct* dependence between variables appearing in an expression and the variable on the left-hand side of

the assignment (except for the case of assignments of the form $Y := Y$). For instance x directly depends on y and z in the statement $x = y + z$;. When variables are unchanged by the command we call it *propagation*. Propagation only happens when a variable is not affected by the command, not when it is copied from another variable. If the variable is set to a constant, we call this a *reinitialization*.

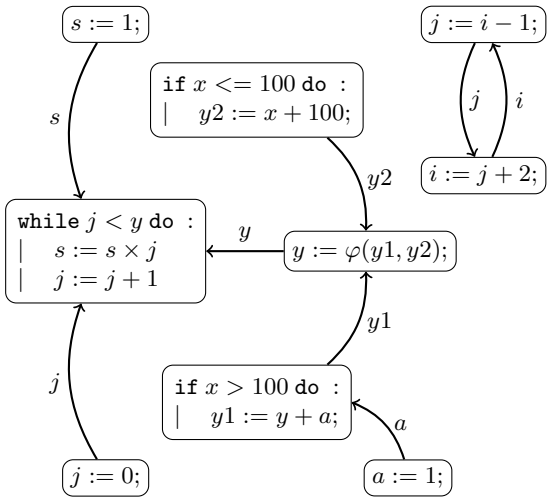
The quasi-invariance comes with an *invariance degree* which is the number of time the loop needs to be peeled to be able to hoist the corresponding invariant. We can then implement program transformations that reduce the overall complexity while preserving the semantics.

3.1 Invariance Degree

First, we want to warn the reader that the wanted transformation on a `WHILE` program may bring a renaming issue if a peeled conditional chunk changes the value of a reused variable (we give an intuition of that in section 4.1). Then, to simplify and be able to show an interesting example, here we introduce the *φ-function* that, at runtime, can choose the correct value of a variable depending on the path just taken. Note that this issue does not exist on a SSA form (see subsection 4.2).

```
i := 0;
x := 42;
y := 5;
a := 12;
while i < 100 do :
|   j := 0;
|   s := 1;
|   while j < y do :
|   |   s := s × j;
|   |   j := j + 1;
|   if x > 100 do :
|   |   y1 := x + a;
|   if x <= 100 do :
|   |   y2 := x + 100;
|   y := φ(y1, y2);
|   j := i − 1;
|   a := 1;
|   i := j + 2;
```

(a) An example



(b) Dependency graph of the outer `while` loop

Fig. 4: Exemple of dependency graph and invariance degrees

We now consider a loop $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \ldots, \mathtt{C_n}]$. We will build a *dependence graph* $\mathrm{Dep}(\mathtt{C})$ from the information given by the DFGs.

First, we need to introduce some notations. Given a variable i, we define the set $i^{\prec}$ as $\{\mathtt{C_k} \mid i \in \mathrm{Out}(\mathtt{C_k})\}$. Given a command $\mathtt{C_m}$ and a variable $i \in \mathrm{In}(\mathtt{C_m})$, we denote as ${\mathtt{C_m}}^{\prec i}$ the subset of $i^{\prec}$ defined as follows:

– it is an initial segment of $i^{\prec}$ w.r.t. the order $<_m$ on $\{C_k \mid k = 1, \dots n\}$ defined as

$$\mathtt{C_{m-1}} <_m \mathtt{C_{m-2}} <_m \cdots <_m \mathtt{C_1} <_m \mathtt{C_n} <_m \mathtt{C_{n-1}} <_m \cdots <_m \mathtt{C_m};$$

– a command $\mathtt{C_k}$ is the largest element of $\mathtt{C_m}^{\prec i}$ w.r.t. $<_m$ if and only if it is neither a `while` nor a `if`.

This defines the subset of *principal dependences* of the command $\mathtt{C_m}$ w.r.t. a given variable i. Intuitively, this principal dependence is the last command preceding $\mathtt{C_m}$ which modified the value of the variable i. However, since `while` and `if` commands may be skipped, we have to consider several main dependences in general. Based on this, we can build the *dependence graph* which simply consists in writing the principal dependences of each command.

Definition 6 *Let* $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \dots, \mathtt{C_n}]$ *be a command. We define the directed graph* $\mathrm{Dep}(\mathtt{C})$ *as follows:*
– *the set of vertices* $V^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\{\mathtt{C_1}, \dots, \mathtt{C_n}\}$*;*
– *the set of edges* $E^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\uplus_{m=1}^{n} \uplus_{i \in \mathrm{In}(\mathtt{C_m})} \mathtt{C_m}^{\prec i}$*;*
– *the source* $s(i)$ *of the edge* $\mathtt{C_k} \in \mathtt{C_m}^{\prec i}$ *is* $\mathtt{C_k}$*;*
– *the target* $t(i)$ *of the edge* $\mathtt{C_k} \in \mathtt{C_m}^{\prec i}$ *is* $\mathtt{C_m}$*.*

The *invariance degree* $\deg_{\mathtt{C}}(\mathtt{C_m})$ of a command $\mathtt{C_m}$ w.r.t. $\mathtt{C}$ is then defined as follows. When clear, we will avoid writing the subscript $\mathtt{C}$ to ease notations. If $\mathtt{C_m}$ is a source in $\mathrm{Dep}(\mathtt{C})$, then $\deg(\mathtt{C_m}) = 1$. If $\mathtt{C_m}$ has a reflective edge in $\mathrm{Dep}(\mathtt{C})$, then $\deg(\mathtt{C_m}) = \infty$. Otherwise, we write $\mathrm{Fib}(\mathtt{C_m})$ the set of vertices in $\mathrm{Dep}(\mathtt{C})$ defined as $\{\mathtt{C_k} \mid \exists e \in E^{\mathrm{Dep}(\mathtt{C})}, s(e) = \mathtt{C_k}, t(e) = \mathtt{C_m}\}$, and define $\deg(\mathtt{C_m})$ by the following equation, where $\chi_{>m}(i) = 1$ if $i > m$ and $\chi_{>m}(i) = 0$ otherwise:

$$\deg(\mathtt{C_m}) = \max\left(\{\deg(\mathtt{C_i}) + \chi_{>m}(i) \mid \mathtt{C_i} \in \mathrm{Fib}(\mathtt{C_m})\}\right)$$

In particular, if $\mathtt{C_m}$ is part of a cycle in $\mathrm{Dep}(\mathtt{C})$, its degree is equal to ∞.

We now define, for all $i \in \mathbf{N} \cup \{\infty\}$, the sets

$$\deg^{-1}(\mathtt{i}) = \{\mathtt{C_k} \mid \deg(\mathtt{C})_k = i\} \qquad \deg_{+}^{-1}(\mathtt{i}) = \{\mathtt{C_k} \mid \deg(\mathtt{C_k}) \geqslant i\},$$

and we note $\mathrm{maxdeg}(\mathtt{C})$ the smallest integer such that $\deg_{+}^{-1}(\mathrm{maxdeg}(\mathtt{C})) = \deg^{-1}(\infty)$. The following lemma will be used in the proof of the main theorem.

Lemma 1. *Consider the set* $\deg^{-1}(\mathtt{i})$ *for an integer* $i > 0$ *and the relation induced from the dependency graph, i.e.* $\mathtt{C_i} \to \mathtt{C_j}$ *if and only if there is a sequence of edges from* $\mathtt{C_i}$ *to* $\mathtt{C_j}$ *in* $\mathrm{Dep}(\mathtt{C})$*. Then* $(\deg^{-1}(\mathtt{i}), \to)$ *is a partial order.*

Proof. It is clear that $\to$ is transitive and reflexive. We only need to show that it is antisymmetric. I.e. that there are no two commands $\mathtt{C_i}, \mathtt{C_j}$ such that $\mathtt{C_i} \to \mathtt{C_j}$ and $\mathtt{C_j} \to \mathtt{C_i}$. We suppose that two such commands can be found and show it leads to a contradiction. Indeed, if such a situation arises, it means that the dependency graph contains a cycle $\mathtt{P_1}, \dots, \mathtt{P_k}$ with $\mathtt{P_1} = \mathtt{P_k} = \mathtt{C_i}$. By definition of the degree, one has $\deg(\mathtt{P_{i+1}}) \geqslant \deg(\mathtt{P_i})$. More to the point, one has $\deg(\mathtt{P_{i+1}}) > \deg(\mathtt{P_i})$ as long as $\mathtt{P_i} = \mathtt{C_k}$ and $\mathtt{P_{i+1}} = \mathtt{C_h}$ with $k > h$. Now, it is clear that one of the inequalities $\deg(\mathtt{P_{i+1}}) \geqslant \deg(\mathtt{P_i})$ has to be strict, as no sequence $\mathtt{C_{i_1}}, \mathtt{C_{i_2}}, \dots, \mathtt{C_{i_k}}$ with $i_1 < i_2 < \cdots < i_k$ can form a cycle. This implies that $\deg(\mathtt{C_i}) > \deg(\mathtt{C_i})$; a contradiction.

Based on the invariance degree, we will be able to *peel* loops. For this purpose, we define the following notation. Given a sequence of commands $[\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n]$, we write $[\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$ the subsequence in which all commands of degree strictly less than i are removed. We then define $\mathtt{if}^i = \mathtt{if}\ \mathtt{E}\ \mathtt{then}\ [\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$, and $\mathtt{while}^i = \mathtt{while}\ \mathtt{E}\ \mathtt{then}\ [\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$. We can now state the main theorem of the paper, which provides a general framework for *peeling* loops.

Theorem 1. *Let* $\mathtt{C} := \mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1; \mathtt{C}_2; \ldots, \mathtt{C}_n]$ *be a command. Then*

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

The proof of this theorem is based on an induction, using the following lemma.

Lemma 2 *Let* $\mathtt{C} := \mathtt{while}\ \mathtt{E}\ \mathtt{do}\ [\mathtt{C}_1; \mathtt{C}_2; \ldots, \mathtt{C}_n]$*, and* $\mathtt{D} := \mathtt{if}^1; \mathtt{while}^2$*. Then* $[\![\mathtt{C}]\!] \equiv [\![\mathtt{D}]\!]$ *and for each command* $\mathtt{C}_m$ *appearing in* $\mathtt{while}^2$*,* $\deg_{\mathtt{C}}(\mathtt{C}_m) = \deg_{\mathtt{D}}(\mathtt{C}_m) + 1$.

Proof. We start by proving the claim that for each command $\mathtt{C}_m$ appearing in $\mathtt{while}^2$, $\deg_{\mathtt{C}}(\mathtt{C}_m) = \deg_{\mathtt{D}}(\mathtt{C}_m) + 1$. This is a consequence of the fact that the dependency graph $\mathrm{Dep}(\mathtt{D})$ is obtained from $\mathrm{Dep}(\mathtt{C})$ by removing all vertices $\mathtt{C}_m$ for commands $\mathtt{C}_m$ of degree 1, together with their outgoing edges. Note that this defines a well-formed graph since by definition of the degree, a command of degree 1 may only depend on commands which are themselves of degree 1 (i.e. edges of target $\mathtt{C}_m$ are removed as well). Now, it is clear from the definition of the dependence degree that $\deg_{\mathtt{C}}(\mathtt{C}_m) = \infty$ implies $\deg_{\mathtt{D}}(\mathtt{C}_m) = \infty$, and that if $\deg_{\mathtt{C}}(\mathtt{C}_m)) = d$ the command $\mathtt{C}_m$ only depended commands of degree at most d. From section 1 we can prove by induction that $\deg_{\mathtt{D}}(\mathtt{C}_m) = d - 1$.

Then, one should realise that commands of degree 1 are in fact invariants of the loop $\mathtt{C}$. It is then clear that $[\![\mathtt{C}]\!] \equiv [\![\mathtt{D}]\!]$.

4 In practice

In the previous section, we have seen that the transformation is possible from and to a WHILE-language. This section will progressively show that we can do it in real languages by introducing our implementations. First it will present our *proof of concept*[8] which does both analysis and propose a transformation from `C` to `C`. After we will explain how we implemented a prototype analysis in a real compiler.

4.1 Proof of concept (PoC)

To easily and quickly integrate our transformation, we decided to use "pycparser"[9], a `C` parser written in `Python`. The principal interest was to simply get and manipulate an Abstract Syntax Tree. Using a "WhileVisitor" we list all nested `while`-loops, then, with a bottom-up strategy (the inner loop first), this tool analyses and transforms the code if an invariant or quasi-invariant is detected. The analysis is divided in two parts: the DFG construction and the invariance degree computation.

[8] `https://github.com/ThomasRuby/LQICM_On_C_Toy_Parser`
[9] `https://github.com/eliben/pycparser`

Analysis The first part aims to list relations between statements. In this implementation we decided to define a relation object by one list of pairs (for the direct dependencies) and two sets (for the propagations and reinitializations) of variables. A relation is computed for each command using a top-down strategy following the dominance tree. The relations are composed when the corresponding command is a sequence of commands. As described previously, we compute the correction and the maximum relations possible for a `while` or `if` statement. With those relations, we compute an invariance degree for each statement in the loop regarding to the relations listed (algorithm 1)

Peeling loops from `C` to `C` On a non-SSA form (see subsection 4.2), variables are often reused to store temporary values. The problem is that if we hoist a part of loop which changes the value of one of those variables it's possible to change the semantic. Furthermore, it's harder if those variables are modified in a conditional chunk, in this case a φ*-function* is needed. This issue is illustrated in Figure 5 if we replace `y1`, `y2` by `y` and φ*-function* is removed.

Implementation details For this PoC we decided to not consider hard renaming (with φ*-function*) for the simple reason that it's not necessary in real compiler. It is however able to peel `C` programs in SSA form with no invariant conditional statements (see the examples of the repository).

This implementation is almost 400 lines of `Python`. It is able to compute relations of each commands or sequence of commands. This tool focuses on a restricted `C` syntax and considers all functions as non-pure. Functions with side effects can be seen as an anchor in the sequence of statements, commands can not be moved around. But we can restrain the conditions for peeling. We can allow to hoist pure functions as in [13]. All other side effects can be broken by this transformation, and thus should not be moved.

4.2 Prototype pass in LLVM

Compilers, and especially LLVM on which we are working, use an *Intermediate Representation* (IR) to handle programs. This is a typed assembly-like language that is used during all the stages of the compilation. Programs (in various different languages) are first translated into the IR, then several optimizations are performed (implemented in so-called *passes*), and finally the resulting IR is translated again in actual assembly language depending on the machine it will run on. Using a common IR allows to do the same optimizations on several different source languages and for several different target architectures.

One important feature of the LLVM IR is the *Single Static Assignment* form (SSA). A program is in SSA form if each variable is assigned at most once. In other words, setting a program in SSA form require a massive α-conversion of all the variables to ensure uniqueness of names. The advantages are obvious since this removes any name-aliasing problem and ease analysis and transformation.

The main drawback of SSA comes when several different paths in the Control Flow reach the same point (typically, after a conditional). Then, the values used after this point may come from any branch and this cannot be statically decided. For example, if

the original program is `if (y) then x:=0 else x:=1;C`, it is relatively easy to turn it into a pseudo-SSA form by α-converting the `x`: `if (y) then` x_0`:=0 else` x_1`:=1;C` but we do not know in `C` which of x_0 or x_1 should be used.

SSA solves this problem by using *φ-functions*. That is, the correct SSA form will be `if (y) then` x_0`:=0 else` x_1`:=1;` $X:=\varphi(x_0, x_1)$`; C`.

While the SSA itself eases the analysis, we do have to take into account the φ functions and handle them correctly.

Existing. LLVM does have Loop Invariant Code Motion (LICM) pass which hoists direct invariants out of loops. Used with unrolling and instruction combination optimizations it can "peel" quasi-invariants. However, as far as we know, it does not compute invariance degrees and does not detect quasi-invariant chunks.

Preliminaries First, we want to visit all loops using a bottom-up strategy (the inner loop first). Then, as for the LICM, our pass is derived from the basic `LoopPass`. Which means that each time a loop is encountered, our analysis is performed.

At this point, the purpose is to gather the relations of all instructions in the loop to compose them and provide the final relation for the entire loop.

Then a `Relation` is generated for each command using a top-down strategy following the dominance tree. The *SSA* form helps us to gather dependence information on instructions. By visiting operands of each assignment, it's easy to build our map of `Relation`. With all the current loop's relations gathered, we compute the compositions, condition corrections and the maximums relations possible as described in section 2.2. Obviously this method can be enhanced by an analysis on bounds around conditional and number of iterations for a loop. Finally, with those composed relations we compute an invariance degree for each statement in the loop following algorithm 1.

This algorithm is dynamic. It stores progressively each degree needed to compute the current one and reuse them. Note that, for the initialization part, we are using LLVM methods (`canSinkOrHoist`, `isGuaranteedToExecute` etc...) to figure out if an instruction is movable or not. These methods provide the anchors instructions for the current loop.

Peeling loop idea The transformation will consist in creating as many basic blocks before the loop as needed to remove all quasi-invariants. For each block created, we include every commands with a higher or equal invariance degree. For instance, the first `preheader` block will contain every commands with an invariance degree higher or equal to 1, the second one, higher or equal to 2 etc... to maxdeg. The final loop will contain every commands with an invariance degree equal to ∞.

Of course, hoisting quasi-invariant of high degrees is not necessarily a good idea since it requires peeling the loop many times and thus greatly increase the size of the code. The decision to hoist or not will depend on the quasi-invariance degree, the size of the loop, ...

```
Data: Dependency Graph and Dominance Graph
Result: List of invariance degree for each command
Initialize degrees of use to ∞ and others to 0;
for each command C_m do
    if the current degree deg(C_m) ≠ 0 then
        return deg(C_m);
    else
        Initialize the current degree deg(C_m) to ∞;
        if there is no dependence for the current chunk then
            deg(C_m) = 1;
        else
            for each dependent command ordered (subsection 3.1) compute the degree deg(C_d) do
                if deg(C_d) = ∞ or C_d = C_m then
                    return ∞;
                end
                if deg(C_m) ≤ deg(C_d) and d > m then
                    deg(C_m) = deg(C_d) + 1;
                else
                    deg(C_m) = deg(C_d);
                end
            end
        end
        return deg(C_m)
    end
end
```

Algorithm 1: Invariance degree computation.

Implementation details The only chunks considered in the current implementation are the one consisting of `while` (any loops in LCSSA form) or `if-then-else` (any forks which have a common post dominator existing in the loop) statements.

This implementation (including a preliminary version of peeling) is almost 3000 lines of `C++`. It is able to compute relations of each commands or sequence of commands. However, it has, for the moment, some restrictions on the form of the loop analyzed. First, loops with several exit blocks are ignored and left intact (typically a loop including a `break`); furthermore, this tool considers all functions as non-pure as for the Proof of Concept. Even with these restrictions, the pass is able to optimise code that was previously left untouched, thus illustrating the power of the method.

5 Conclusion and Future work

5.1 Results

Developers expect that compilers provide certain more or less "obvious" optimizations. When peeling is possible, that often means: either the code was generated; or the developers prefer this form (for readability reasons) and expect that it will be optimized by the compiler; or the developers haven't seen the possible optimization (mainly because of the obfuscation level of a given code).

Our generic pass is able to provide a reusable abstract dependency graph and the quasi-invariance degrees for further loop optimization or analysis.

In this example (Figure 5), we compute the same factorial several times. We can detect it statically, so the compiler has to optimize it at least in `-O3`. Our tests showed that is done neither in LLVM nor in GCC (we also tried `-fpeel_loops` with profiling). The generated assembly shows the factorial computation in the inner loop.

```
0  int main(){
1     int n,i,x,y,z,a;
2     i=0;
3     x=42;
4     y=5;
5     a=12;
6     while(i<n){
7        j=0; // 1
8        s=1; // 1
9        while (j<y) { // 3
10          s=s*j;
11          j=j+1;
12       }
13       if (x>100) { // 2
14          y1=x+a;
15       }
16       if (x<=100) { // 1
17          y2=x+100;
18       }
19       y:=φ(y1,y2);
20       a=1; // 1
21       j=i-1; // ∞
22       i=j+2; // ∞
23    }
24    return i;
25 }
26
27 Dependencies:
28 [[], [], [0, 1, 4, 3], [5], [], [], [7], [6]]
29
30 Invariance degrees:
31 [1, 1, 3, 2, 1, 1, ∞, ∞]
```

peeling

```
0  int main() {
1    int n, i, x, y, z, a;
2    i=0;
3    x=42;
4    y=5;
5    a=12;
6    if(i<n){ // 1
7      j=0;
8      j_1=j;
9      s=1;
10     s_1=s;
11     [While]
12     [If1]
13     [If2]
14     y2_1=y2;
15     y:=φ(y1,y2);
16     a=1;
17     j=i-1;
18     i=j+2;
19   }
20   if(i<n){ // 2
21     j=j_1;
22     s=s_1;
23     [While]
24     [If1]
25     y1_1=y1;
26     y2=y2_2;
27     y:=φ(y1,y2);
28     j=i-1;
29     i=j+2;
30   }
31   if(i<n){ // 3
32     [While]
33     y1=y1_1;
34     y:=φ(y1,y2);
35     j=i-1;
36     i=j+2;
37   }
38   while(i<n){ // ∞
39     j=i-1;
40     i=j+2;
41   }
42   return i;
43 }
```

Fig. 5: Hoisting inner loop

Moreover, the computation time of this kind of algorithm compiled with `clang` in `-O3` still computes n times the inner loop so the computation time is quadratic, while hoisting it result in linear time. For the example shown in Figure 5 (LLVM-IR in Figure 6a), our pass will compute the right degrees.

```
1  ...
2  while.cond:
3    %j.0 = phi i32 [ 0, %entry ], [ %add20, %while.end ]
4    %y.0 = phi i32 [ 5, %entry ], [ %y.2, %while.end ]
5    %i.0 = phi i32 [ undef, %entry ], [ %j.0, %while.end ]
6    %a.0 = phi i32 [ 5, %entry ], [ 0, %while.end ]
7    %cmp = icmp slt i32 %j.0, %rem
8    br i1 %cmp, label %while.cond5, label %while.end21
9
10 while.cond5:
11   %fact.0 = phi i32 [ %mul, %while.body8 ], [ 1, %while.cond ]
12   %i.1 = phi i32 [ %add, %while.body8 ], [ 1, %while.cond ]
13   %cmp6 = icmp sle i32 %i.1, %y.0
14   br i1 %cmp6, label %while.body8, label %while.end
15
16 while.body8:
17   %mul = mul nsw i32 %fact.0, %i.1
18   %add = add nsw i32 %i.1, 1
19   br label %while.cond5
20
21 while.end:
22   %cmp10 = icmp sgt i32 %rem3, 100
23   %add12 = add nsw i32 %rem3, %a.0
24   %add12.y.0 = select i1 %cmp10, i32 %add12, i32 %y.0
25   %cmp14 = icmp sle i32 %rem3, 100
26   %add17 = add nsw i32 %rem3, 100
27   %y.2 = select i1 %cmp14, i32 %add17, i32 %add12.y.0
28   %add20 = add nsw i32 %j.0, 1
29   br label %while.cond
30
31 while.end21:
32   ret i32 %i.0
33 ...
```

(a) LLVM Intermediate Representation

```
--- vim -O1 EarlyOpt 7m5,276s
3808 number of loops
20465 number of instructions hoisted by LICM
2266 number of loops with several exit blocks
125 number of loops not well formed
2391 sum of loop not analyzed
1417 sum of loop analyzed by LQICM
2476 number of quasi-invariants detected
335 number of quasi-invariants Chunk detected
23 Number of chunks with deg >= 2
0.0686567 Average
29 Number of deg >= 2
0.0117124 Average

--- emacs -O1 EarlyOpt 15m52,556s
3161 number of loops
16415 number of instructions hoisted by LICM
1775 number of loops with several exit blocks
150 number of loops not well formed
1925 sum of loop not analyzed
1236 sum of loop analyzed by LQICM
2197 number of quasi-invariants detected
311 number of quasi-invariants Chunk detected
35 Number of chunks with deg >= 2
0.11254 Average
50 Number of deg >= 2
0.0227583 Average
```

(b) Statistics on vim and emacs.

To each instruction printed corresponds an invariance degree. The assignment instructions are listed by loops, the inner loop (starting with `while.cond5`) and the outer loop (starting with `while.cond`). The inner loop has its own invariance degree equal to 3 (line `10`). Remark that we do consider the `phi` initialization instructions of an inner loop. Here `%fact.0` and `%i.1` are reinitialized in the inner loop condition block. So `phi` instructions are analyzed in two different cases: to compute the relation of the current loop or to give the initialization of a variable sent to an inner loop. Our analysis only takes the relevant operand regarding to the current case and do not consider others.

Statistics have been generated by our pass on the two editors `vim` and `emacs` to evaluate the magnitude of new optimizations possible (Figure 6b). Note that the result change a lot regarding to when our pass is called. Here, it is placed before all loop optimizations to compare with number of instructions detected by LICM. However, it's important to understand that quasi-invariant could be generated by other passes.

The code of this pass is available online[10]. To provide some real benchmarks on large programs we need to finish the transformation. We are currently working on.

[10] `https://github.com/ThomasRuby/lqicm_pass`

5.2 Further works

The pass is currently a prototype. The transformation is still in preliminary form and even the analysis is making some approximations (*e.g.* considering all functions as non-pure) that hamper its efficiency. We will obviously work further on the pass to finish the transformation and increase the number of cases we can handle.

On a more theoretical side, the current analysis is strongly inspired by other ICC analysis such as *Size Change Termination* [11] (from which the Data Flow Graphs and Multipaths are taken) or the mwp-analysis (from which the loop correction idea is taken) [9]. As shown in subsection 2.1, it is easy to adapt the method to similar analysis, and most of the existing code can be reused. Thus, we plan on implementing a mwp-inspired complexity analysis in LLVM, which should be able to guarantee the polynomiality of large parts of the code.

Such certificates built at compile-time can be used in a *Proof Carrying Code* paradigm. If the compiler is trusted (for example, untrusted developers upload source-code onto an applications store and the compilation is made by the trusted store), then the certificate ensure that certain properties (in this case, complexity) are valid.

Acknowledgments The authors wish to thank L. Kristiansen for communicating a manuscript [8] that initiated the present work. Jean-Yves Moyen is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 655222 - Walgo; Thomas Rubiano is supported by the ANR project "Elica" ANR-14-CE25-0005; Thomas Seiller is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 659920 - ReACT.

References

1. A. Abel and T. Altenkirch. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(1), 2002.
2. P. Baillot and K. Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 201(1), 2009.
3. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2, 1992.
4. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *CLMPS*. 1962.
5. John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7), 1970.
6. J.-Y. Girard. Linear logic. *Th. Comp. Sci.*, 50, 1987.
7. M. Hofmann. Linear types and Non-Size Increasing polynomial time computation. In *LICS*, pages 464–473, 1999.
8. L. Kristiansen. Notes on code motion. manuscript.
9. L. Kristiansen and N. D. Jones. The flow of data and the complexity of algorithms. *Trans. Comp. Logic*, 10(3), 2009.
10. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, 1981.
11. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL*, 2001.
12. Jean-Yves Moyen. Resource control graphs. *ACM Trans. Computational Logic*, 10, 2009.
13. Litong Song, Yoshihiko Futurama, Robert Glück, and Zhenjiang Hu. A loop optimization technique based on quasi-invariance. 2000.