Implementing the mwp-flow analysis

Neea Rusch
Augusta University

Collaborators:
C. Aubert, T. Rubiano, T. Seiller

15 November 2021

Introduction

- Our research focuses on static program analysis of imperative programs
- Using a technique inspired by implicit computational complexity
- This talk will demonstrate how to use this technique to analyze variable value growth
- We have modified, extended and made this technique practical with a working protype

Outline

1. Preliminaries:
   - Implicit compulational complexity
   - Static analysis
2. Theoretical foundation: *mwp*-analysis
3. Implemented analysis
4. Other applications & future plans

Computational complexity

- Computational complexity evaluates resource usage of programs, usually in terms of space and time
- Given some decision problem and a specific machine model: how much resources are needed to solve the problem?
- Resource usage is expressed in terms of input: more resources are allowed as input size grows
- Decision problems can then be classfied into different complexity classes
- Polynomial (P) class represents problems that are feasible

Implicit computational complexity (ICC)

- Implicit approach has no machine model: restrict language instead
- Ability to represent program in the restricted syntax ensures P bounds
- There are many approaches to ICC
- Our technique is based on "Copenhagen school" method of data flow analysis

Static analysis

- Static analysis enable programmer to analyze program repeatedly
- Analysis performed on source code without executing the program
- Analysis can evaluate different properties, e.g. error checks, running time, data flow
- There are many ways to implement based on requirements: abstract interpretation, data flow analysis, etc.

Static analysis of complexity

- Complexity analysis focuses on analysing running time or memory usage
- There are two natural parts: termination analysis and data size analysis

Static analysis of complexity

Relevant considerations:

1. Precision: interested in single or multiple complexity classes; existence of bounds or tight bounds?
2. Source code language: imperative, declarative, specific source code?
3. Automation: does program need annotations?

Alternative approaches

Name	Language	Focus
SPEED	C++	time bounds
ComplexityParser	Java	polytime complexity
COSTA	Java Bytecode	cost and termination
RaML	OCaml	resource usage, time
pymwp	C (subset)	value size growth

Theoretical foundation: *mwp* analysis

- 2008 paper by Neil Jones and Lars Kristiansen:
  "A Flow Calculus of mwp-Bounds for Complexity Analysis"
- This technique is related in spirit to abstract interpretation as it bounds *transitions* between states (commands), instead of states
- "Careful and detailed analysis of the relationship between resource requirements of computation and the way data might flow during computation"

Syntax

Variable	$\mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots$
Expression	`X` | `e + e` | `e * e`
Boolean Exp.	`e = e`, `e < e`, etc.
Commands	`skip` | `X := e` | `C;C` | `loop X {C}` | `if b then C else C` | `while b do {C}`

mwp Calculus

Analyze variable value growth by:

1. Assigning a vector to each variable
2. Collecting vectors into a matrix
3. Applying derivation rules to evaluate program complexity

Flows represent quantitative information of variables on each other:

0	no dependency
m	maximal
w	weak polynomial
p	polynomial

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix} \qquad \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix} \tag{E1}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix} \tag{E3}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{X2 = X1 + X2} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \tag{A}$$

Example

```
loop X3 {X2 = X1 + X2}
```

$$\texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix} \tag{L}$$

Non-determinism & failure

Jones & Kristiansen wanted to be able to analyze as many programs as possible:

- implemented non-deterministic derivation rules
- up to 3 rules can be applied to expressions
- single program can have multiple matrices
  (program of n lines can have up to 3^n derivations)
- if program analysis cannot be completed, stop and explore a different strategy

Open questions

The original *mwp*-analysis was theoretical

There were open questions:

1. Can it be applied to richer languages?
2. How powerful and convenient is this technique?

Implementing *mwp* analysis

Two significant modifications were needed to enable implementation:

1. Non-determinism of original analysis was impractical: replaced by deterministic derivation rules

$$\texttt{X2 = X1 + X1} : \begin{pmatrix} m & w(0,0)\ p(1,0)\ w(2,0) \\ 0 & 0 \end{pmatrix}$$

- All derivations are represented in the same matrix

Implementing *mwp* analysis

Two significant modifications were needed to enable implementation:

2. Changing handing of failure: introduced a new flow ∞ to represent failure locally

$$0, m, w, p, \infty$$

- Enables completing every derivation
- Provides fine-grained infromation on source of failure on programs that do not have polynomially bounded growth

Prototype: `pymwp`

- Implementation of *mwp*-analysis on a subset of C99, in Python
- Open source: `github.com/statycc/pymwp`
- If analysis succeeds:
  - program uses at most a polynomial amount of space
  - if it terminates, it will do so in polynomial time
- If variable grows too much, polynomial bound cannot be guaranteed

Resolving practical inefficiencies

Representing all derivations in 1 matrix leads to exponential growth in matrix

This issue was resolved with 2 strategies:

1. decoupling computation by using *delta graph*
2. compositionality enables reusing results

Resolving practical inefficiencies

Delta graph enables decoupling computation of *existence* of bounds and computing its values

- Delta graph tracks all derivation branches that end in infinite value
- Whenever a subtree cannot be completed, simplify the graph
- If no branches remain, analysis cannot be completed
- If at least one branch remains, it is possible to compute actual bounds

Resolving practical inefficiencies

Compositionality of analysis enables computing result once then reusing the result it in the future

- Analysis can be performed on *parts* of source code
- It is possible to analyze a function, then save the result
- Previously analyzed result can be reused at next execution
- Expensive computation needs to be carried out once

Results

Our implementation demonstrates *mwp*-analysis is:

- **Programming language-independent**: reason abstractly about imperative languages and apply to real languages
- **Compositional**: analyze parts of code once and reuse as needed, unlike many other static analysis methods
- **Modular**: same theory can be applied to different problems after changes in internal machinery
- **Abstracted**: ICC influenced technique abstracts problems with intervals, value ranges, iterations, etc.
- **Extendable**: Modifications of internal mechanism may enable capturing tight bounds, other complexity classes, etc.

Other applications & future plans

The following work has been completed so far:

1. Loop optimization: using dependency analysis borrowed from ICC to detect inefficiencies in loops and to optimize them, integrated with LLVM (published)
2. `pymwp` standalone static analyzer, for analyzing variable value growth, for subset of C code (submitted)

Other applications & future plans

Future directions for complexity analysis include compiler integration:

1. Leverage intermediate representation
2. Static single assignment (SSA) form for efficiency and fine-grained information
3. Certified complexity analysis to be able to integrate with CompCert

Other applications & future plans

mwp-analysis is an innovative way to capture dependecies.

It can be used to solve many other problems:

1. Loop parallelisation (currently in progress)
2. Extend loop optimization to integrate with CompCert (future plan)
3. Floating-point analysis to track growth of error in precision (long-term plan)
4. ...

Other applications & future plans

4. …maybe you have some ideas?

What would you do with *mwp* flow analysis?