Programming Language Design and Implementation

Introduction

Winter Semester 2017/18; 11 October, 2017

Thomas Noll et al.
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1718/pldi/

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Outline

Overview

Programming Language Design and Implementation

Focus of ACM PLDI conference

Design, implementation, theory, applications, and performance of programming languages and related systems

Overview

Programming Language Design and Implementation

Focus of ACM PLDI conference

Design, implementation, theory, applications, and performance of programming languages and related systems

Research directions

- New and well motivated theoretical results
- Inter-disciplinary work which connects programming languages with other fields
- Concepts and generalizations that lead to better understanding of current developments
- [Thorough experimental studies which result in new insights]
- [Well designed systems that solve a difficult practical challenge]

Areas Covered in this Seminar

Areas

- Compilation
  - *Compiler Construction* (SS 2017)
  - *Semantics and Verification of Software* (SS 2015)
- Analysis
  - *Static Program Analysis* (WS 2016/17)
  - *Compiler Construction* (SS 2017)
- Verification
  - *Semantics and Verification of Software* (SS 2015, now)
  - *Introduction to Model Checking* (SS 2016)

Outline

Aims of this Seminar

Goals

Aims of this seminar

- Independent understanding of a scientific topic
- Acquiring, reading and understanding scientific literature
- Writing of your own report on this topic
- Oral presentation of your results

Aims of this Seminar

Requirements on Report

Your report

- Independent writing of a report of 10–15 pages
- Complete set of references to all consulted literature
- Correct citation of important literature
- Plagiarism: taking text blocks (from literature or web) without source indication causes immediate exclusion from this seminar
- Font size 12pt with “standard” page layout
- Language: German or English
- We expect the correct usage of spelling and grammar
  - $\geq$ 10 errors per page $\Longrightarrow$ abortion of correction
- LaTeX template will be made available on seminar web page

Requirements on Talk

Your talk

- Talk of 30 minutes
- Available: projector, presenter, [laptop]
- Focus your talk on the audience
- Descriptive slides:
  - $\leq$ 15 lines of text
  - use (base) colors in a useful manner
  - number your slides
- Language: German or English
- No spelling mistakes please!
- Finish in time. Overtime is bad
- Ask for questions
- Have backup slides ready for expected questions
- LaTeX template will be made available on seminar web page

Outline

Important Dates

Deadlines

- 13 November: Detailed outline of report due
- 11 December: Full report due
- 15 January: Presentation slides due
- 30 January (?): Seminar

Important Dates

Deadlines

- 13 November: Detailed outline of report due
- 11 December: Full report due
- 15 January: Presentation slides due
- 30 January (?): Seminar

Missing a deadline causes immediate exclusion from the seminar

Important Dates

Selecting Your Topic

Procedure

- You obtain(ed) a list of topics of this seminar.
- Indicate the preference of your topics (first, second, third).
- Return sheet by Monday (16 October) via e-mail/to secretary.
- We do our best to find an adequate topic-student assignment.
  - disclaimer: no guarantee for an optimal solution
- Assignment will be published on web site next week.
- Then also your supervisor will be indicated.

Selecting Your Topic

Procedure

- You obtain(ed) a list of topics of this seminar.
- Indicate the preference of your topics (first, second, third).
- Return sheet by Monday (16 October) via e-mail/to secretary.
- We do our best to find an adequate topic-student assignment.
  - disclaimer: no guarantee for an optimal solution
- Assignment will be published on web site next week.
- Then also your supervisor will be indicated.

Withdrawal

- You have up to three weeks to refrain from participating in this seminar.
- Later cancellation (by you or by us) causes a not passed for this seminar and reduces your (three) possibilities by one.

Outline

1: Adaptive LL(*) parsing

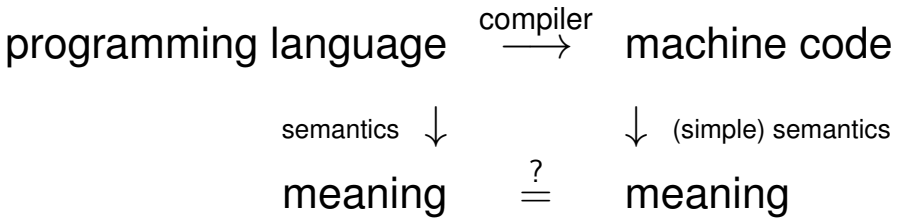
- Compiler Construction course: top-down ($LL(k)$) analysis
- Grammar analysis done at parser construction time (lookahead sets)
- Idea: move grammar analysis to parsing time ($ALL(*)$)
- Can handle any non-left-recursive context-free grammar
- Time complexity $O(n^4)$ but consistent linear runtime on grammars used in practice
- Supported by ANTLR 4 parser generator

2: Counterexamples for bottom-up parsing conflicts

- Compiler Construction course: bottom-up ($LR(k)$) analysis
- shift/reduce and reduce/reduce conflicts hard to understand
- Approach: generation of compact, helpful counterexamples for *LALR* grammars that demonstrate conflict (e.g., ambiguity)

3: Verified compilation

- Compiler correctness:



- Challenges: shared-memory interaction, concurrency, separate compilation for C-like languages
- New proof technique: logical simulation relations
- Supports compositional reasoning
- Formalized in Coq theorem prover

4: Compilation of functional languages

- Central concept: join points for re-combining control flow
- Example:

```
if (if e1 then e2 else e3) then e4 else e5
```

  rearranged to:

```
if e1 then (if e2 then e4 else e5)
       else (if e3 then e4 else e5)
```

  optimised by join points (texttj4 and `j5`):

```
let { j4 () = e4; j5 () = e5 }
in if e1 then (if e2 then j4 () else j5 ())
          else (if e3 then j4 () else j5 ())
```

- Often indirectly treated as functions/continuations (“functional GOTOs”)
- Idea: add join points to functional intermediate language
- Allows new optimizations to be performed
- Implemented in Glasgow Haskell Compiler

5: Synthesis of machine code from semantics

- Basis: semantic specification of straight line machine code given as a Quantifier-Free Bit-Vector (QFBV) logic formula
- Automated synthesis of instruction sequences based on semantic specifications of single instructions
- Employs efficient strategies to handle search space (divide-and-conquer, pruning)

6: Improving accuracy for floating-point expressions

- Problem: rounding errors in floating point arithmetic
- State of the art: numerical methods applied to mitigate rounding error
- Requires manually rearranging expressions and understanding floating point arithmetic in detail
- Here: tool support to automatically discover accuracy improvements
- Heuristic search to estimate and localize rounding errors
- Database of rules to generate improvements

7: Program enumeration for compiler testing

- Approach: program viewed as syntactic structure P (syntactic skeleton) parametrised by identifiers V (variable names)
- Skeletal program enumeration (SPE) problem: given P and V, enumerate a set of programs exhibiting all possible variable usage patterns within P
- Paper proposes effective realization of SPE
- Enables rigorous compiler testing by exploiting three important observations:
  - different variable usage patterns trigger various compiler optimization passes
    - Example:

```
int a,b=1;          int a,b=1;          int a,b=1;
b = b-a;            a = b-b;            a = b-b;
if(a)               if(a)               if(b)
  a = a-b;            a = a-b;            a = b-b;
...                 ...                 ...
// a uninitialised  // a = 0 at line 3   // b = 1, a = 0
                    // (dead code elimination)  // (constant folding)
```

  - most real compiler bugs were revealed by small P
  - SPE is exhaustive w.r.t. P and V
- Thus: level of guarantee superior to most existing compiler testing techniques

Outline

8: Points-to analysis

- Points-to analysis: class of pointer analysis algorithms to detect null pointer references and aliasing effects
- Essential approach: allocation-site abstraction to model heap objects
- Here: novel heap abstraction for object-oriented languages by finite automata that represent type-consistent objects
- Efficient points-to analysis enabled by merging equivalent automata

9: Polymorphic type checking

- Addresses performance penalty of parametric polymorphism
- Example of polymorphic function:

```
bTwice :: forall a: Bool -> a -> (a -> a) -> a
bTwice b x f = if b then f (f x) else x
```

- New variant: levity polymorphism
- Allows abstractions over memory layout (function calling conventions)
- Identifies restrictions that are necessary in order to compile levity-polymorphic functions

10: String Analysis for Vulnerability Detection and Repair

- Common source for security vulnerabilities in web applications: string manipulation errors in input validation and sanitization code
- Here: string analysis techniques to automatically identify and repair vulnerabilities
- Procedure:
  1. extraction of client- and server-side input validation and sanitization functions
  2. modelling as deterministic finite automata
  3. identification of potential errors by comparing with manually specified attack patterns or by searching for inconsistencies between client and server-side
  4. application of automated repair techniques

11: Static Detection of DoS Vulnerabilities

- Algorithmic complexity attack: malicious party exploits worst-case behaviour of algorithm to cause denial-of-service
- Prominent example: regular expression denial-of-service (ReDoS)
- Attacker provides a carefully-crafted input string that triggers worst-case behaviour of the matching algorithm
  - e.g., exponential blow-up due to powerset construction for determinisation
- Here: technique for automatically finding ReDoS vulnerabilities in programs
  - identifies vulnerable regular expressions in program
  - checks whether “evil” input string can be matched against a vulnerable regular expression

12: Static analysis of performance bugs

- Aim: avoid redundant traversal bugs
- Arise if program repeatedly iterates over data structure without intermediate modification
- Thus computations can be memoized and re-used across loop iterations instead
- Here: formalisation and novel static analysis for automatic detection of such bugs

13: Analysing undefined behaviour of C programs

- Introduces “negative” semantics of C language
  - gives meaning to correct programs
  - rejects undefined programs
- Example:

```
for (int i = 0; i <= N; ++i) { ... }
```

  - undefined behaviour caused by signed overflow
    $\Longrightarrow$ compiler may assume `N` + 1 iterations
  - for `unsigned int i`: overflow defined
    $\Longrightarrow$ compiler would now need to consider non-terminating case (`N` = `UINT_MAX`)
- Extraction of undefinedness checker from formal semantics
- Evaluation on benchmark test suite for undefined behaviour

Outline

14: Verification of information-flow security

- Goal: protecting confidentiality of information manipulated by a computing system
- Formally verification that end-to-end behaviour satisfies information-flow policies
- Challenge: preservation of properties through compilation and cross-language linking
- New concept: observation function to cover
  - policy specification
  - state indistinguishability
  - execution behaviour

15: Cartesian Hoare logic for verifying k-safety properties

- Classical safety properties: absence of “bad” program traces
  - examples: no null pointer dereference, no deadlock, mutual exclusion
- k-safety properties: absence of a “bad” interaction between k traces
  - example: determinism
  - $\forall x, y : x = y \implies f(x) = f(y)$
  - requires two execution traces to establish violation
- Introduction of Cartesian Hoare Logic (CHL) for verifying k-safety properties
- Automated verification algorithm and tool implementation

16: Model checking concurrent programs

- Problem: state space explosion due to interleaving of concurrent programs
  - Example processes:

    ```
    p: write x;    q: write x;    r: read x;
    ```

  - classical view: all accesses mutually dependent
    $\Longrightarrow$ 6 executions to consider
  - but: `r` only thread that reads `x`
    $\Longrightarrow$ `p.q.r` equivalent to `q.r.p`, `p.r.q` to `q.p.r`, and `r.q.p` to `r.q.p`.
    $\Longrightarrow$ only 3 executions to consider
- Approach: establish maximal causality reduction (MCR) to explore state-space with (provably) minimal number of executions
- Each execution represents maximal set of causally equivalent executions
- Scales very well due to parallelisability

17: Verifying pointer programs using separation logic

- Separation logic: extension of Hoare logic for modular reasoning about pointer programs

$$\text{(seq)}\ \frac{\{A\}\ c_1\ \{C\} \qquad \{C\}\ c_2\ \{B\}}{\{A\}\ c_1; c_2\ \{B\}}$$

- Novel features: separating conjunction and separating implication to support reasoning about disjoint heap parts

$$\text{(frame)}\ \frac{\{A\}\ c\ \{B\}}{\{A * F\}\ c\ \{B * F\}} \qquad \text{if } \mathrm{write}(c) \cap \mathrm{free}(F) = \emptyset$$

- Here: proof system for full separation logic
- Developed in sequent calculus style as set of inference rules for manipulating heap structures

Outline

Some Final Hints

Hints

- Take your time to understand your literature.
- Be proactive! Look for additional literature and information.
- Discuss the content of your report with other students.
- Be proactive! Contact your supervisor on time.
- Prepare the meeting(s) with your supervisor.
- Forget the idea that you can prepare a talk in a day or two.

Some Final Hints

Hints

- Take your time to understand your literature.
- Be proactive! Look for additional literature and information.
- Discuss the content of your report with other students.
- Be proactive! Contact your supervisor on time.
- Prepare the meeting(s) with your supervisor.
- Forget the idea that you can prepare a talk in a day or two.

We wish you success and look forward to an enjoyable and high-quality seminar!