Bachelor Project Proposals 2022

Logic and Semantics & Programming Languages Groups

General information

The projects in this specialisation aim at reasoning about *programming* in a scientific manner. The underlying motivation is to increase the reliability of software systems, in particular for parallel and distributed systems. This is achieved by using better programming languages, writing better code, and proving its properties (using logic or algorithms).

The topics are often related (but not exclusively) to the following BSc courses that you know:

- Programming Café (Magnus Madsen)
- Computability and Logic (Jaco van de Pol)
- Programming Languages (Anders Møller)
- Compilers (Aslan Askarov, Amin Timany)

Some projects are more theoretical in nature (logic, semantics of programming languages, correctness proofs, foundations), while other projects are quite practical (compiler technology, tool support, analysis algorithms, test strategies, applications, computer security).
The teachers are experts in Logic and Semantics and Programming Languages, and are all highly committed to provide dedicated and intensive supervision to their project groups:

- Aslan Askarov (language-based security, compilation)
- Lars Birkedal (program semantics, program verification, concurrency)
- Magnus Madsen (programming languages, compilers, type and effect systems)
- Anders Møller (programming languages, program analysis)
- Andreas Pavlogiannis (software analysis, concurrency, algorithmic verification)
- Jean Pichon-Pharabod (concurrency, semantics)
- Jaco van de Pol (model checking, automata)
- Bas Spitters (type theory, blockchain, computer aided cryptography)
- Amin Timany (program verification, type theory)

Please contact the supervisor directly, and as soon as possible, when you are interested in one of the projects on the following pages.
For general questions, or in case of doubt, you can always contact pavlogiannis@cs.au.dk. He can also help you to find a supervisor if you have other ideas for a bachelor project subject.

Magnus Madsen (magnusm@cs.au.dk)

Auto-Completion and Program Synthesis

Integrated development environments (IDEs), such as Intellij IDEA, aid programmers by providing auto-completion for field and method names. Dependently-typed programming languages such as Agda and Idris take this a step further by allowing the programmer to have the compiler fill in entire expressions based on the types of the program. Essentially, the compiler searches for expressions that satisfy the requirements (e.g. types) of a hole in the program.

The aim of this project is to: (1) explore the design space of such program completions, and to (2) design and implement such a system for the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language being developed at Aarhus University (see flix.dev).

Compiler Fuzzing

Compilers are large and complex pieces of software. The correctness of a compiler is paramount: A compiler that silent mis-compiles (i.e. wrongly translates) a program is dangerous: We cannot trust the programs we compile and run! We can test compilers by writing unit tests, but unfortunately such tests tend to only test the 'happy path' of the compiler. Moreover, the amount of unit tests that can be written are limited. Instead, compiler fuzzing techniques have been proposed. A compiler fuzzy typically takes a test suite as input and subtly changes the programs in a systematic fashion and then re-compiles the test suite. This process is fully automatic and can be run for hours. Often such techniques, with suitably clever 'mutation strategies', are able to find significantly more bugs than those found by unit testing.

The aim of this project is to: (1) explore the design space of compiler fuzzing techniques, and to (2) design and implement such a system for the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language being developed at Aarhus University (see flix.dev).

Inlining - The Holy Grail of Compiler Optimizations

A compiler checks an input program for well-formedness and translates it to a lower-level format, i.e. JVM bytecode or machine code. An *optimizing* compiler aims to not just translate a program, but to make it run faster. In particular, high-level programs often contain a lot of abstracts that when compiled naive are a source of overhead. Inlining is one of the most powerful program optimization techniques. During inlining, function definitions are expanded to their body expressions which often enable further optimizations. However, inlining also increases the size of the generated code, and can lead to non-termination if applied carelessly.

The aim of this project is to: (1) explore the design space of compiler optimization strategies for functional and imperative programming languages, and to (2) design and implement such optimizations for the Flix programming language. This includes looking into techniques such as monomorphization, closure elimination, case-of-case optimizations, partial evaluation, and more.

Delimited Continuations

Most programming languages support a notion of exceptions. An exception is a special control-flow construct that allows a function to abort execution and to transfer control to an exception handler. Exceptions are typically implemented as an intrinsic part of the programming language. However, it is possible and useful to offer more powerful features that enable users or library authors to implement their own control-structures. Delimited continuations is one such feature. In a programming language with delimited continuations, exceptions can be implemented entirely as a library. Moreover, delimited continuations can enable resumable exceptions and other interesting control-flow constructs.

The aim of this project is to: (1) explore the design space of delimited continuations and to (2) design and implement a notion of delimited continuations for the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language being developed at Aarhus University (see flix.dev).

Termination Analysis

A common programming mistake is to unintentionally write an infinite loop. Most contemporary programming languages such as C, C++, C#, Java, Kotlin, Scala, etc. do not help programmers avoid such issues. Termination analysis or termination checkers describe a wide-range of techniques that can be used to verify that a program (or part of a program) always terminates, for any input. These techniques range from sophisticated type systems to the use of SMT solvers. In the case of functional programming, such termination checkers may try to ensure termination by verifying that recursion is always on structurally smaller elements.

The aim of this project is to explore the design space of termination analysis for a functional language. The work will include reading papers, language design, and implementation in a real-world programming language being developed at Aarhus University (see flix.dev).

Andreas Pavlogiannis (pavlogiannis@cs.au.dk)

Predictive Analysis of Concurrent Programs

Concurrent and distributed programs are notoriously hard to write correctly because of the non-determinism in scheduling. They even make simple testing painful — you run your program and it crashes, you run it again in order to debug it and the bug has disappeared! Predictive analyses are program analysis techniques that aim exactly at that: they take as input correct program executions, and try to predict the existence of other, buggy executions.

The project aims to develop new techniques for predicting bugs in concurrent programs. It is based on new algorithms that offer theoretical improvements over previous approaches. The starting point will be implementing these algorithms on a Dynamic Analysis Tool called Rapid, in an efficient and performant way. Based on the insights obtained in this process, there is room for developing new, better predictive algorithms.

The project requires interest in concurrent/distributed systems, program analysis and algorithms.

Relevant sources:
https://drops.dagstuhl.de/opus/volltexte/2021/14393/pdf/LIPIcs-CONCUR-2021-16.pdf
https://github.com/umangm/rapid

Data Structures for Logical Times in Distributed Systems

Time in distributed systems is a relative measure, as different processes execute at different speeds. Instead of using absolute time to infer the order in which certain events happen in the system, we typically use logical times. The idea goes back to Lamport's timestamps, and is used extensively in modern distributed systems and the web. Logical times are currently computed using a simple data structure known as a vector clock.

The project will explore the replacement of vector clocks by a new data structure, called tree clocks. Tree clocks were recently introduced for solving a different problem, namely for the analysis of concurrent programs, but have the potential to also improve logical timing. The project aims at realizing this potential, by extending vector clocks to the distributed setting, implementing them and testing them

The project requires interest in distributed systems, algorithms.

Relevant sources:
https://en.wikipedia.org/wiki/Vector_clock?wprov=sfti1

Dynamic Language Graph Reachability

Graph reachability is one of the most fundamental problems on graphs: given a graph G and two nodes u, v is there a path from u to v? Language reachability takes as additional input a language L over an alphabet A, and the edges of G are labeled with letters from A. The language reachability problem asks for a path from u to v that moreover produces a string s along its edges that belongs to L. This problem has wide applications in formal language theory, databases and program analysis.

The project will study the language reachability problem under dynamic updates, i.e., the graph is changing over time by introducing and removing edges. From a program-analysis perspective, the graph is an abstraction of the program, and dynamic updates correspond to changes in the model while the program is being developed. The aim of the project is to implement some new algorithms for this task, and test them in practice.

The project requires interest in automata, formal language theory, graph algorithms, program analysis.

Relevant sources:
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.8958&rep=rep1&type=pdf
https://dl.acm.org/doi/10.1145/2491956.2462159

Algorithmic Analysis of Bidirected Vector Addition Systems

Vector addition systems with states (VASS) are finite automata that manipulate one or more counters in each transition. They are used to model a wide range of real systems, where the counters may correspond to resources like energy consumption or utilities accumulated while performing certain tasks. As such, they have received a lot of attention in both theoretical and practical research. Unfortunately, their analysis is often computationally intractable.

The project targets a simpler sub-class of VASS called Bidirected VASS, and aims to develop new algorithms for their analysis. It will be based on insights taken from very recent papers in the literature. The project is mostly theoretically centered, but there is also opportunity for implementing the new algorithms and testing them.

The project requires interest in automata, formal language theory, algorithms.

Relevant sources:
https://en.wikipedia.org/wiki/Vector_addition_system?wprov=sfti1
https://cs.au.dk/~pavlogiannis/publications/papers/popl22.pdf

Andersen's Pointer Analysis

(together with Anders Møller)

"Given a program with pointers, what memory locations may some pointer point to during execution?". This is a program analysis question that becomes very useful when reasoning about program correctness. Andersen's Pointer Analysis (APA) is a standard, effective technique for predicting pointer information statically (i.e., without executing the program).

The project aims to develop efficient variants of the basic APA algorithm. Potential directions include the development and evaluation of various heuristics, as well as a parallel version of APA, in order to improve its scalability.

The project requires interest in parallel processing, graph theory, program analysis.

Sources:
https://dl.acm.org/doi/10.1145/3434315

Lars Birdekal (birkedal@cs.au.dk)

Polymorphic Type Inference

Type systems play a fundamental role in programming languages, both in practise and in theory.

A type system for a programming language can be understood as a syntactic discipline for maintaining levels of abstraction. For example, types can be used to enforce that clients of a module cannot get access to the data representation used in the module. A sound type system ensures that there are no runtime errors when a well-typed program is run.

For some programming languages and type systems it is possible to automatically (algorithmically) infer types, thus relieving the programmer of the burden of annotating program phrases with types. A type inference algorithm is thus an algorithm which infers types. A type inference algorithm can be thought of as a kind of static analysis: the type inference algorithm analyses the program statically, and infers which invariants (described by types) the program maintains.

There are many type systems and type inference algorithms. A very successful type system is Milner's polymorphic type system and type inference algorithm for this type system include algorithms based on unification and on constraint-solving.

The aim of this project is to learn about type systems and type inference. Specifically, the initial goal of the project is to learn about Milner's polymorphic type system and inference algorithms based on unification. The core part of the project includes implementing polymorphic type inference. Several extensions are possible, both of more theoretical nature and of more implementation-oriented nature, depending on student interest

Literature

- Tofte, M: ML Type Inference, Algorithm W. Chapter 2 of his PhD thesis titled: Operational Semantics and Polymorphic Type Inference.
- Sestoft, P: Programming Language Concepts, Ch. 6

Jean Pichon

Verified compilation of LLVM IR to WebAssembly (co-advisor Bas Spitters)

WebAssembly is a language designed to be the compilation target to run programs in web browsers. WebAssembly is interesting, because it is at the same time a widely used web standard, supported by all major browser vendors, and a small and clean enough language to work with in a proof assistant.
LLVM IR is a high-level assembly language that is used as the internal representation of the optimisation phases of the LLVM suite of compiler tools. Compilers for a variety of languages, including C (clang) and Rust (rustc), are composed of a front-end compiling the language to LLVM IR, and the LLVM optimisers and backends. A large subset of LLVM IR has also been formalised in a proof assistant.
The goal of the project is to (1) explore compilation between mainstream languages, and (2) explore verified compilation.

Note: The main difference between the two languages is the type of control structure, see below; this project would assume that you are given a structured control overlay.

Two Mechanisations of WebAssembly 1.0, Watt et al.
https://hal.archives-ouvertes.fr/hal-03353748/

Modular, Compositional, and Executable Formal Semantics for LLVM IR, Zakowski et al.
https://www.seas.upenn.edu/~euisuny/paper/vir.pdf

Formal verification of a realistic compiler, Leroy
https://xavierleroy.org/publi/compcert-CACM.pdf

Relooper

WebAssembly is an unusual compilation target, because it only features structured control ("if", "while", etc.), and not unstructured control ("goto"). This means that to compile a (more typical) low-level language with unstructured control to WebAssembly, one needs to reconstruct an equivalent program with structured control. The goal of the project is to (1) explore how such a structured control reconstruction algorithm, like Relooper or Stackifier, works, and (2) explore how to use a proof assistant to prove correctness of algorithms.

https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2

Type-checking exception level integrity in instruction set architectures

The definition of a modern instruction set architecture (ISA) like Arm or RISCV covers thousands of instructions, and the definition of one instruction can be hundreds of lines of code in a domain-specific language like Sail. The size of these definitions means that they are likely to contain errors. One particularly important property of an ISA definition is the integrity of exception levels (aka "protection rings" or "domains"): a userland program should not be able to affect operating systems private registers, and an operating system should not be able to affect hypervisor private registers. The goal of this project is to explore how to design a type system to identify violations of exception level integrity, and to explore how to implement a typechecker so that it scales to mainstream ISAs.

https://github.com/rems-project/sail

Language-Based Information-Flow Security, Sabelfeld and Myers
https://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf

Jaco van de Pol

Hammer Away: Integrated Automated and Interactive Reasoning

With Jaco van de Pol (jaco@cs.au.dk) and Simon Wimmer (swimmer@cs.au.dk)

SMT solvers (like Z3 [1] or CVC 4 [2]) can automatically prove and disprove mathematical propositions and as such have greatly influenced the state-of-the-art in automated analysis of software.

Full verification, i.e. proving the correctness of software, often relies on an interactive process within a so-called interactive theorem prover. Success stories in this area include verified compilers and a verified operating system [3,4].

So-called hammers [3] have greatly changed the degree of automation in this process by incorporating SMT solvers. However, they usually work in a "one-shot-manner".

A number of already proved facts are collected in the interactive theorem prover (ITP), combined with a proposition to be proved, and translated to the SMT solver's lingua franca.

The whole bundle is then given to the SMT solver and one either receives a proof or an answer "don't know".

The goal of this work is to integrate ITPs (such as Coq, Isabelle/HOL, HOL 4, and Lean) with SMT solvers more tightly. To this end, we want to implement a semi-decision procedure for the verification of recursive functions on top of existing hammer infrastructure. This would mostly be useful for finding counterexamples (similar to QuickCheck) but we also want to study how far this can assist in finding proofs (time permitting).

This work would include significant implementation work in the ML programming language to extend the Isabelle/HOL theorem prover [5]. There is already a lot of ground work given by the Sledgehammer infrastructure in Isabelle/HOL and an existing implementation of the algorithm in OCaml and Scala. There will however also be many interesting research questions to answer, e.g. how to handle higher-order functions and algebraic datatypes properly. The work should be concluded by an empirical evaluation of the effectiveness of the procedure compared to current approaches for finding counterexamples in Isabelle/HOL.

This project should be a great fit for students that have already taken interest in functional programming and/or theorem proving and who want to strengthen their skills, working on a real code base.

Literature

[1] https://github.com/Z3Prover/z3/wiki
[2] https://cvc4.github.io/
[3] https://arxiv.org/abs/2003.06458
[4] https://sel4.systems/
[5] https://isabelle.in.tum.de/

Strongly Connected Components with Binary Decision Diagrams

With Andreas Pavlogiannis and Jaco van de Pol

Computing the strongly connected components (SCCs) on a directed graph is a fundamental operation that lies at the heart of many applications, including program analysis and model checking. However, in model checking the graphs represent the state spaces of programs/systems, which makes them huge. Exploiting the structure of the program, they are effectively compressible in a "symbolic" way using data structures like binary decision diagrams (BDDs) [1].

The project requires interest in computability and logic, graphs, automata, programming

The project has multiple directions, possibly pursued by multiple BSc groups. Both directions involve algorithm design, implementation and experiments, and can be extended by theoretical questions about algorithm complexity.

Direction 1: SCC for Coloured graphs
This subproject aims to develop and implement a new, fast algorithm for computing SCCs using BDDs in colored graphs: each vertex has a color, and each SCC must be monochromatic, i.e., only contain vertices of the same color. The project will extend a recently developed algorithm for colored SCC decomposition [2]. An interesting theoretical question is if the algorithm can be sublinear in the number of colours.

Direction 2: SCC with saturation
The most simple SCC algorithm with BDDs has been improved in multiple directions. The skeleton approach [3] provides a clever method to limit the number of BDD operations to linear in the size of the graph. The saturation approach [4] provides a strategy to limit the size of the intermediate BDDs while doing a reachability computation. The goal of the project is to extend the skeleton approach with saturation. A practical question is if the saturation approach is indeed effective in combination with skeletons. A more theoretical question would be if the combined algorithm still runs in the worst case linear number of BDD operations.

Sources:
[1] See Chapter 6:1-3 of Huth & Ryan, "Logic in Computer Science" on BDDs
[2] N. Beneš, L. Brim, S. Pastva, D. Šafránek, Symbolic Coloured SCC Decomposition
[3] R. Gentilini, C. Piazza, A. Policriti, Computing strongly connected components in a linear number of symbolic steps
[4] Y. Zhao and G. Ciardo, Symbolic computation of strongly connected components and fair cycles using saturation

I/O-efficient Binary Decision Diagrams

With Jaco van de Pol: jaco@cs.au.dk and Steffan Sølvsten: soelvsten@cs.au.dk

Binary Decision Diagrams (BDD) [1] form an efficient data-structure to store large sets of state vectors efficiently. BDDs have many applications, like solving combinatorial puzzles (for instance N-queens, tic-tac-toe). A more serious application is to store the huge state space of a model checker efficiently.

BDDs have a few simple operations (like taking the union and intersection) and these operations are usually defined and implemented as recursive functions, working in main memory. However, when the BDD is larger than fits in main memory, it should be stored on disk. The traditional algorithms are extremely inefficient since they do random disk access.

To this end, we have implemented an IO-efficient implementation of the BDD algorithms, called Adiar [2,3]. The implementation builds upon an off-the-shelf IO-efficient library TPIE [4]. This library offers IO-efficient data-structures, like sorted lists and priority queues. The implementation is based on a theory of IO-efficient algorithms [5].

The challenges (potentially for several groups) are:
1. To improve the current algorithms, for instance:
   a) Changing the variable ordering of the BDD
   b) Using partial in-main-memory calculations
2. To design, implement and evaluate other IO-efficient BDD operations
3. To argue the correctness and I/O complexity of such IO-efficient algorithms
4. To construct and analyse combinatorial benchmarks using large BDDs on disk
5. To apply and experiment with I/O-efficient BDDs in a model checker

Literature:
[1] See Chapter 6:1-3 of Huth & Ryan, “Logic in Computer Science” on BDDs
[2] Steffan Sølvsten et al. Efficient Binary Decision Diagram Manipulation in External Memory
[3] Steffan Sølvsten, The Adiar repository on github, especially the future work section
[4] The TPIE library
[5] Lars Arge, The I/O-complexity of Ordered Binary-Decision Diagram manipulation

Bas Spitters

spitters@cs.au.dk

Programming with dependent types

Stop fighting type errors! Type-driven development is an approach to coding that embraces types as the foundation of your code - essentially as built-in documentation your compiler can use to check data relationships and other assumptions. With this approach, you can define specifications early in development and write code that's easy to maintain, test, and extend.
This kind of programming with so-called dependent types is becoming more and more popular; see for example languages like Idris, agda, dependently typed haskell, F*, Coq, ...

In this project you will have several possibilities to explore this programming technique, either from a practical, or a more theoretical side.

https://www.fstar-lang.org/
https://www.manning.com/books/type-driven-development-with-idris
http://adam.chlipala.net/cpdt/

High assurance cryptography

With Bas Spitters and Diego Aranha

High Assurance Cryptography
https://github.com/hacspec/hacspec

Cryptography forms the basis of modern secure communication. However, its implementation often contains bugs. That's why modern browsers and the linux kernel use high assurance cryptography:
one implements cryptography in a language with a precise semantics and proves that the program meets its specification.

In this project, you will write a reference implementation of a number of key cryptographic primitives in a safe subset of rust, while at the same time specifying the implementation. You will use either semi-automatic or interactive tools to prove that the program satisfies the implementation.

There are a number of local companies interested in this technology. So, this work will be grounded in practice.

Property based testing for smart contracts

With Bas Spitters: spitters@cs.au.dk

Smart contracts are small programs deployed on a blockchain that typically capture aspects of real world contracts and allow one to automate (financial) services while avoiding a trusted third party.
Because these contracts control millions of kroner in value, and because they are hard to change when deployed, correctness is imperative. Moreover, very expensive accidents have happened that could have been avoided. E.g.
https://medium.com/blockchain-academy-network/preventing-an-8m-attack-on-ethereums-bzx-defi-platform-with-property-based-testing-12234d9479b7

In this project, you will develop a property based testing framework for the rust smart contract language. Property-based testing is an automated, generative approach to effective software testing, relying on executable properties and (random) test data generators.

This work takes place in the context of the cobra center.
https://cs.au.dk/research/centers/concordium/
https://blockchainacademy.dk/

Amin Timany

Relational Reasoning

with Amin Timany: timany@cs.au.dk

Relational reasoning about programs is a very powerful concept with many applications in the study of programs and programming languages, e.e., expressing compiler correctness, compiler security, etc. A particularly interesting notion of relation between two programs is contextual equivalence. Two programs are contextually equivalent if, as part of a bigger program, each could be used in place of the other without altering the behavior of the whole program. Apart from expressing properties of compilers and programming languages, one can think of contextual equivalence as the gold standard of comparing programs, e.g., when justifying program refactoring. For instance, we may, as a programmer, wish to replace one implementation of a data structure with another, more efficient implementation. In such a case, we can prove that the new implementation is contextually equivalent to the old implementation, and hence that replacing the old implementation with the new one should not change the behavior of the program. A common approach to proving contextual equivalences is to use the logical relations technique.

The aims of this project are as follows:
- Learn the formal concepts of contextual equivalence and logical relations.
- Prove a few interesting cases of program equivalence.

Literature

- Part 3 of the book "Advanced Topics in Types and Programming Languages" by Benjamin Pierce
- An introduction to Logical Relations by Lau Skorstengaard (https://cs.au.dk/~birke/papers/AnIntroductionToLogicalRelations.pdf)

Aslan Askarov (aslan@cs.au.dk)

http://askarov.net

Privacy-preserving browsing

Modern web-pages fetch content from multiple sources, which enables untrusted third parties to fingerprint browsers against user expectations. Browser fingerprinting often goes beyond basic cookie-tracking and relies on device-specific details such as screen size, installed fonts, and the network IP address. This project will study the state-of-the-art techniques for browser fingerprinting and tracking and explore language-based approaches for their confinement.

References

- Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels
  https://hal.inria.fr/hal-01943496v4/document
- Protecting Users by Confining JavaScript with COWL
  https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-stefan.pdf

Security of serverless applications

Serverless applications provide a new methodology for building large scalable applications in the cloud. Application logic is partitioned into a set of functions that execute on demand, often through an API, and which further interact with other components such as object stores, event handlers, and message queues. The heterogeneity of these applications pose interesting security challenges as there is no single runtime that can accurately trace information flows throughout the system. This project will study the architecture of building serverless applications and will focus on a few concrete canonical components of serverless applications, e.g., message queues, with the goal of enriching their API to accurately track information flow within the system.

References

- Valve: Securing Function Workflows on Serverless Computing Platforms
  https://dl.acm.org/doi/fullHtml/10.1145/3366423.3380173
- Secure serverless computing using dynamic information flow control
  https://arxiv.org/pdf/1802.08984.pdf

Mitigating side channels in just-in-time compilers

Just-in-time (JIT) compilers generate code for bytecode-based and dynamic programming languages. To achieve this, JIT compilers often use speculative optimizations which depend on the data handled by the program. Recent work shows that such speculative behavior of JIT compilers can be exploited by adversaries to leak sensitive information. This project will study the principles of designing and implementing a JIT compiler for a simple bytecode language, and explore the methodology for mitigating JIT leaks.

References

- JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation
  https://ieeexplore.ieee.org/document/9152649
- Formally Verified Speculation and Deoptimization in a JIT Compiler
  https://dl.acm.org/doi/pdf/10.1145/3434327