Overview Bachelor Projects 2021

Logic & Semantics and Programming Languages

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)
- Compilation (Aslan Askarov)

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, type systems, program analysis)
- Anders Møller (programming languages, program analysis)
- Andreas Pavlogiannis (software analysis, concurrency, algorithmic verification)
- Jaco van de Pol (model checking, automata)
- Bas Spitters (functional programming, type theory, blockchain)
- 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 jaco@cs.au.dk. He can also help you to find a supervisor if you have other ideas for a bachelor project subject.

Project Overview

1. **Aslan Askarov: compilation and security**
    a. Troupe-related projects
    b. Resource-based side channels in browsers
    c. Anti-fuzzing for Javascript

2. **Lars Birkedal: type theory, program semantics**
    a. Polymorphic Type Inference

3. **Magnus Madsen: programming languages, program analysis, type systems**
    a. Package Management
    b. Auto-Completion and Program Synthesis
    c. Fixpoint Engine
    d. Termination Analysis

4. **Andreas Pavlogiannis: algorithmic verification, software analysis, concurrency**
    a. Dynamic Analysis of Concurrent Programs
    b. Andersen's Static Pointer Analysis
    c. Formal language theory for static program analysis

5. **Jaco van de Pol: model checking and automata**
    a. Parallel graph algorithms for game analysis
    b. Simulator for Timed Automata with Parameters
    c. I/O-efficient Binary Decision Diagrams
    d. Joining a Model Checking Contest

6. **Bas Spitters: Type theory, functional programming, blockchain**
    a. Dependently typed programming
    b. High assurance cryptography
    c. Property based testing for smart contracts

7. **Amin Timany: Program verification, type theory**
    a. Relational Reasoning

1a. Troupe-related projects

With Aslan Askarov: aslan@cs.au.dk

Troupe is a programming language for concurrent and distributed programming with dynamic information flow control. Troupe is a research language intended for experimenting with ideas in the area of information flow control. As of 2020, we use Troupe in Language-Based Security course at AU. A user-guide to the language is available at https://troupe.cs.au.dk. The compiler is written in Haskell, and the runtime is written in Javascript. The source code is open and is accessible on Github.

Possible projects include but are not limited to

- Information-flow—aware optimizations in the backend
- Adding monitoring for integrity security policies (current implementation only supports confidentiality)
- A general mechanism for wrapping around existing JS libraries
- Design and implementation of persistent objects
- Design and implementation of a module system
- Native backend

References

- Troupe project homepage: http://troupe.cs.au.dk

1b. Resource-based side-channels in modern browsers

With Aslan Askarov aslan@cs.au.dk

Modern browsers routinely execute untrusted javascript code. Even when execution is isolated (i.e., in an incognito browsing window), the code still has access to shared hardware resources such as CPU or GPU caches or shares internal data-structures (such as event loops). This access creates dangerous side-channels that may leak private information, such as the identity of a website that is open in a background tab. This project will revisit known attacks such as Loophole (see ref below) in the context of modern browsers (that include mitigations against such attacks) and further explore new resource-based browser attack vectors.

Literature

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/vila

1c. Anti-fuzzing for JavaScript

With Aslan Askarov: aslan@cs.au.dk

Fuzzing is a popular automated testing technique often used to discover vulnerabilities in software in a black-box fashion. Anti-fuzzing is a defense technique that automatically modifies application code to make fuzzing harder, typically by incurring runtime overhead that is negligible in regular use but is significant during fuzzing. In this project, we will explore fuzzing and anti-fuzzing for JavaScript and its security applications via penetration testing (e.g., via a tool called Black Widow).

Literature

https://www.usenix.org/conference/usenixsecurity19/presentation/jung
https://www.cse.chalmers.se/~andrei/bw21.pdf

2a. Polymorphic Type Inference

With Lars Birkedal: birkedal@cs.au.dk

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

3a. Package Management

With Magnus Madsen: magnusm@cs.au.dk

Today, programs are rarely written from scratch, but rather build on a large collection of external libraries. Different languages have different package ecosystems: Java has Maven, JavaScript has NPM, Rust has Cargo, and so forth. All of these languages offer some form of package manager that is used to download, install, upgrade, and keep track of the dependencies of a software project. The problem is non-trivial: For example, how should we handle the situation where a project depends on package A and package B, and A depends on C (version 1.0), but B depends on C (version 2.0)?

The aim of this project is to (1) explore the design space of package managers for programming languages, and to (2) design and implement a package manager 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).

Literature

- Michael Hanus: Semantic Versioning Checking in a Declarative Package Manager
- https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527

3b. Auto-Completion and Program Synthesis

With Magnus Madsen: magnusm@cs.au.dk

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. Effectively, the compiler searches for expressions that satisfy the requirements (e.g. types) of a "hole" in the program. If there are multiple candidates, the compiler uses heuristics (e.g. Machine Learning) to rank the different choices.

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).

Literature

- Edwin Brady: Type Driven Development with Idris

3c. Fixpoint Engine

With Magnus Madsen: magnusm@cs.au.dk

Datalog is a simple, yet powerful, declarative logic programming language. A Datalog program consists of a collection of constraints; each constraint is either a fact or rule. Together, the facts and rules imply a minimal model, a unique solution to any Datalog program. The "fixpoint engine" is the software responsible for computing the solution to a Datalog program. It is essentially a specialized form of database with support for parallel evaluation.

The aim of this project is to: (1) understand how the semi-naive evaluation algorithm works, and to (2) re-design and re-implemented the fixpoint engine in the Flix language to make it faster and easier to maintain.

Literature:

- Stefano Ceri: What you always wanted to know about Datalog (and never dared to ask)
- https://flix.dev/programming-flix/#/fixpoints/

3d. Termination Analysis

With Magnus Madsen: magnusm@cs.au.dk

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).

Literature

- Andreas Abel: Termination Checker for Simple Functional Programs

4a. Dynamic Analysis of Concurrent Programs (multiple projects)

With Andreas Pavlogiannis: pavlogiannis@cs.au.dk

Concurrency occurs when multiple threads interact during the execution of a concurrent program. Concurrent programs are challenging to write correctly, because the programmer must ensure correctness under all possible ways of thread interaction.

Dynamic analyses help detect erroneous program behavior by observing and analyzing concrete program executions instead of the program itself. The simplicity of this approach allows for clever algorithms with low complexity, and there are continuous efforts to get the "best" algorithm for the task.

The purpose of this project is to learn about various correctness notions of concurrent programs (e.g., data-race freedom, deadlock freedom, serializability, atomicity etc.), learn about existing techniques for analyzing such properties, and develop new algorithms for the task. The project has multiple variants (at least 4) that can be taken by different groups, where the focus of each group is on a different property and concurrency model. Each project has a theoretical and an implementation component, however the amount of time allocated in each will be determined based on discussion and personal interests.

Suggested reading

Cormac Flanagan, Stephen N. Freund:
FastTrack: efficient and precise dynamic race detection. Commun. ACM 53(11): 93-101 (2010)

Mohsen Lesani, Todd D. Millstein, Jens Palsberg:
Automatic Atomicity Verification for Clients of Concurrent Data Structures. CAV 2014: 550-567

Yan Cai, Ruijie Meng, Jens Palsberg:
Low-overhead deadlock prediction. ICSE 2020: 1298-1309

Andreas Pavlogiannis:
Fast, sound, and effectively complete dynamic race prediction. Proc. ACM Program. Lang. 4(POPL): 17:1-17:29 (2020)

Umang Mathur, Mahesh Viswanathan:
Atomicity Checking in Linear Time using Vector Clocks. ASPLOS 2020: 183-199

4b. Andersen's Static Pointer Analysis

With Andreas Pavlogiannis: pavlogiannis@cs.au.dk

Pointer analysis is one of the fundamental problems in static program analysis. Given a set of pointers, the task is to produce a useful over-approximation of the memory locations that each pointer may point-to at runtime. The most common formulation is Andersen's Pointer Analysis (APA), defined as an inclusion-based set of pointer constraints.

The purpose of this project is to familiarize with the domain of APA and its usefulness in practice, read literature on existing approaches to APA, and implement some recent algorithms for performing APA. For the last component, the students are expected to contribute their own ideas on top of the existing algorithms, that will make these algorithms fast in practice and hopefully outperform the existing state of the art. The implementation will be followed by experiments on real-world programs.

Suggested reading

Static Program Analysis (Chapter 10)
Anders Møller and Michael I. Schwartzbach

Manu Sridharan, Stephen J. Fink:
The Complexity of Andersen's Analysis in Practice. SAS 2009: 205-221

The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis
A. A. Mathiasen, A. Pavlogiannis

4c. Formal language theory for static program analysis

With Andreas Pavlogiannis: pavlogiannis@cs.au.dk

Static program analyses are clever lightweight scans of the source code that infer useful properties about the program and can detect bugs at an early stage. Various such analyses are cast to the domain of formal language theory, using regular expressions, pushdown automata, etc.

The purpose of this project is to familiarize with the use of formal language theory in static analysis. Several language-theoretic problems in this domain are undecidable. The students will explore variations of such problems, and characterize the cases where the problems remain undecidable vs the ones where decidability is reached. The project has mostly a theoretical flavor, however positive results (i.e., when decidability is achieved) can be implemented and tested in practice.

Suggested Reading

T. Reps. Program analysis via graph reachability. Information and Software Technology, 40(11-12):701–726, 1998.

Thomas W. Reps:
Undecidability of context-sensitive data-independence analysis. ACM Trans. Program. Lang. Syst. 22(1): 162-186 (2000)

Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, Hong Mei. Summary-Based Context- Sensitive Data-Dependence Analysis in Presence of Callbacks. POPL 2015: 83-95.

5a. Parallel graph algorithms for game analysis

With Jaco van de Pol: jaco@cs.au.dk

Model checking is an exhaustive technique to traverse and check the state space (graph) of a concurrent system. Its strength is that it can analyse all possible interleavings of concurrent threads. Its weakness is that the state space of a realistic system is huge. Parallel algorithms are used to speed up the analysis on multi-core processors or GPUs.

The challenge of this project is to study parallel graph algorithms that are used in model checking. Typical examples are parallel algorithms to detect the strongly connected components of a graph, or parallel algorithms to detect accepting cycles in an automaton.

We are also interested in graphs that model *games*, that is: systems with two (or more) players (agents). This has many applications, for instance in control synthesis “system versus environment”, test generation “system versus tester”, or security analysis “defender versus attacker”. The goal is to analyse if one player has a winning strategy, i.e. a procedure to win the game no matter what the other player does.

Possible tasks (sufficient for several projects) could be:

1. Describe existing algorithms for game strategy synthesis (sequential or parallel)
2. Analyse and compare the efficiency and speedup of the parallel algorithms.
3. Analyse and prove the correctness of these algorithms.
4. Design and analyse your own parallel algorithm for game synthesis.
5. Implement your algorithm (e.g. on a cluster, multi-core machine, GPU, …).
6. Collect a diverse benchmark of large graphs (of existing or synthetic graph games).
7. Measure the efficiency and speedup on this benchmark of large graphs.

Literature:

- Marsland, T. Anthony, and Fred Popowich: "Parallel game-tree search." Pattern Analysis and Machine Intelligence, IEEE Transactions on 4 (1985): 442-452.
- Vincent Bloemen, Alfons Laarman, Jaco van de Pol: Multi-core on-the-fly SCC decomposition. In: Principles and Practice of Parallel Programming, PPoPP 2016.
- Andreas Dalsgaard, et al.: A Distributed Fixed-Point Algorithm for Extended Dependency Graphs. Fundamenta Informaticae 161(4): 351-381 (2018)

These references could just form a starting point for your literature search; there are more papers and possible directions on the topic.

5b. Simulator for Timed Automata with Parameters

With Jaco van de Pol: jaco@cs.au.dk

Parametric Timed Automata consist of (networks of) *automata* extended with real time clocks and parameters. The clocks are used to describe real-time properties of embedded software. The parameters are needed because exact timing values are often unknown at design time. We actually want to compute the values of the parameters, for which the system is correct.

The tool Imitator (https://www.imitator.fr/) can compute correct parameters for many interesting properties. It operates on a clever representation of real time values, using polyhedra (or just: systems of inequalities). These algorithms have many built-in heuristics.

The goal of the project is to extend Imitator with a simulator, to visualise the working of the automata and the analysis algorithms, so that it becomes easier to study their efficiency and pinpoint bottlenecks.

Subtasks:

1. Understand the basic theory of Parametric Timed Automata (and polyhedra)
2. Understand the code base of Imitator (which is written in OCaml)
3. Build a basic simulator to navigate the state space computed by Imitator
4. Provide simulation/visualisation aids to understand more complicated algorithms

Literature:

- Imitator toolset (OCaml) and documentation (https://www.imitator.fr/)
- You could look at the UPPAAL toolset for Timed Automata (without parameters) (https://www.uppaal.org)
- Étienne André. IMITATOR II: A Tool for Solving the Good Parameters Problem in Timed Automata. In Yu-Fang Chen and Ahmed Rezine (eds.), INFINITY'10, Electronic Proceedings in Theoretical Computer Science 39, pages 91-99, 2010
- Hoang Gia Nguyen, Laure Petrucci, Jaco van de Pol: Layered and Collecting NDFS with Subsumption for Parametric Timed Automata. ICECCS 2018: 1-9

5c. 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) form an efficient data-structure to store large sets of sequences efficiently. Simply speaking, these diagrams consist of a tree of “decision nodes” which can be thought of as “if-then-else”. The conciseness stems from the fact that equivalent subtrees can be shared (i.e., the diagram is an acyclic directed graph, rather than a tree). 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, using the IO-efficient library TPIE. This library offers IO-efficient data-structures, like sorted lists and priority queues. The implementation is based on a theory of IO-efficient algorithms.

The challenges (for several groups) are:

1. To design other BDD operations in an IO-efficient manner
2. To implement and evaluate IO-efficient BDD operations
3. To argue the correctness of the IO-efficient algorithms
4. To analyse the IO-complexity (number of reads and writes) of the algorithms
5. To construct benchmarks and applications using large BDDs
6. To experimentally evaluate the efficiency of IO-efficient BDD computing

Literature:

- See Chapter 6:1-3 of Huth & Ryan, “Logic in Computer Science” for a quick intro on BDDs
- Henrik Reif Andersen, An Introduction to Binary Decision Diagrams
- Lars Arge, The I/O-complexity of Ordered Binary-Decision Diagram manipulation
- Steffan Sølvsten, The COOM project on github, especially the future work section

5d. Joining a Model Checking Contest

With Jaco van de Pol: jaco@cs.au.dk

Model checking is an automatic procedure to check the required properties of a finite system model. Model checkers usually come with efficient implementations of clever graph algorithms. The properties can be simple reachability queries, or specifications in temporal logic. The models are often provided by a number of parallel automata (processes).

Several model checking contents have been organised, in order to compare the efficiency and applicability of model checkers. Each contest comes with its own format for the model, and typically, a number of categories for various kind of properties.

Several sub-projects are conceivable, both theoretically and practically:

1. Translate the input format for a competition to an existing model checker
2. Improve an existing model checker for a particular category
3. Generate certificates, to double check the correctness of a model checker

Literature:

- See Chapter 3 of Huth & Ryan, “Verification by Model Checking”
- The annual Model Checking Competition, https://mcc.lip6.fr/
- The annual RERS challenge, http://rers-challenge.org/
  (in particular the parallel problems)
- LTSmin high-performance model checker, https://github.com/utwente-fmt/ltsmin
- Simon Wimmer and Peter Lammich, Verified Model-Checking of Timed Automata

6a Programming with dependent types

With Bas Spitters: spitters@cs.au.dk

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/

6b High assurance cryptography

With Bas Spitters: spitters@cs.au.dk 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.

6c 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. You will obtain early access to the concordium blockchain testnet to experiment with your work on real smart contracts.
https://cs.au.dk/research/centers/concordium/
https://blockchainacademy.dk/

7a. 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.g., 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.

Defining and using logical relations (to show program equivalences) for expressive programming languages can be very intricate. However, recently, the development of the Iris program logic, a framework for reasoning about programs, has allowed us to define logical relations for expressive programming languages at a high level of abstraction that are vastly easier to define and use. Moreover, the Iris program logic allows us to prove program equivalences in the Coq proof assistant; a computer tool used to write machine checked proofs.

The main goals of this project are as follows:

- Learning the formal concepts of contextual equivalence and logical relations
- Learning the relevant parts of the Coq proof assistant and the Iris program logic
- Using the existing definition of logical relations in Iris to prove equivalence of interesting programs, e.g., two different stack implementations

Literature:

- Chapter 7 of the book "Advanced Topics in Types and Programming Languages" by Benjamin Pierce
- Iris lecture notes: https://iris-project.org/tutorial-material.html
- Volume 1 of software foundations: https://softwarefoundations.cis.upenn.edu/