# Bachelor Project Proposals 2023

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 committed to 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 (algorithms in program verification, concurrency)
- Jean Pichon-Pharabod (concurrency, semantics)
- Jaco van de Pol (model checking, automata, verification of algorithms)
- 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.

## Aslan Askarov (aslan@cs.au.dk)

http://askarov.net

### Property-based testing in Troupe

Property-based testing is a software testing methodology in which test inputs are automatically generated from high-level specifications. This approach originates from Haskell's QuickCheck library; by now, many programming languages have comprehensive property-based testing libraries.

In this project, you will design and implement a property-based testing library for Troupe – a research programming language for secure distributed and concurrent programming. A characteristic feature of Troupe is its runtime security monitoring, which allows Troupe programs to execute untrusted code securely. However, security monitoring introduces new failure paths corresponding to potential security violations that  are not exhibited in traditional languages. Because of this extensive surface failure, Troupe programs must be well-tested. Property-based testing will significantly boost programmer productivity.

This project will include the following steps:

- learning foundations of property-based testing
- learning foundations of information flow control and how they work in Troupe
- implementing a property-based testing library in Troupe,
- showing how the new library is used by developing a broad suite of specifications for the core Troupe functionality and/or designing a new Troupe library, such as a dictionary or a distributed key/value store
- further extensions to the library based on the experience from the previous step.

Further reading:

- Original QuickCheck paper: https://dl.acm.org/doi/pdf/10.1145/351240.351266
- Troupe website at AU: http://troupe.cs.au.dk

## 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

## Magnus Madsen (magnusm@cs.au.dk)

### Code Formatting

A significant part of compiler implementation focuses on the parser: the compiler phase that turns source code text into abstract syntax trees. However, the opposite direction is often forgotten: the code formatter that turns abstract syntax trees back into neatly formatted source code text. Today, programming languages like Go, IDEs like Intellij IDEA, and linters all come with built-in support for code formatting.

This project aims to explore different strategies for code formatters and experiment with the implementation of such formatters. This project is genuinely part art and part science.

The project offers the opportunity to read research papers, work on language design, and work on a real-world programming language developed at Aarhus University and by a community of open-source contributors (see www.flix.dev).

### Sub-Typing and Sub-Effecting

A type system characterizes the values of an expression, whereas an effect system characterizes the computational side effects of an expression. Programming languages with subtyping allow a more specific type to be used where a less precise type is required (e.g., passing a Student object where a Person object is expected). Similarly, sub-effecting allows a function with a more specific effect to be passed where a less precise effect is expected.

This project aims to explore different strategies for designing and implementing sub-effecting in a programming language focusing on type and effect inference.

The project offers the opportunity to read research papers, work on language design, and work on a real-world programming language developed at Aarhus University and by a community of open-source contributors (see www.flix.dev).

### Tail Recursion Modulo Cons

Tail Recursion Modulo Cons (TRMC) is an essential optimization for functional programming languages that enable very efficient compilation of common functions such as map and filter. The key idea is the compilation of recursive functions into imperative while-loops that operate on mutable data (even though the functional program operates on immutable data).

This project aims to explore different strategies for the implementation of tail calls and specifically for Tail Recursion Modulo Cons (TRMC).

The project offers the opportunity to read research papers, work on language design, and work on a real-world programming language developed at Aarhus University and by a community of open-source contributors (see www.flix.dev).

## Termination Analysis

A common programming mistake is to write an infinite loop. Unfortunately, most contemporary programming languages, such as C, C++, C#, Java, Kotlin, and Scala, do not help programmers avoid such issues. Termination analysis describes a wide range of techniques that can verify that a program (or part of a program) always terminates. For example, by checking that recursive calls always operate on structurally smaller elements.

This project aims to explore strategies for termination analysis in a functional programming language.

The project offers the opportunity to read research papers, work on language design, and work on a real-world programming language developed at Aarhus University and by a community of open-source contributors (see www.flix.dev).

## Andreas Pavlogiannis (pavlogiannis@cs.au.dk)

### Predictive Analyses 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 is based on recent 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 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
https://dl.acm.org/doi/10.1145/3503222.3507734

## 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

(co-advisor Aslan Askarov)

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

### Mechanized Verification and Optimization of Symbolic Graph Algorithms

With Andreas Pavlogiannis and Jaco van de Pol

Graph algorithms are important workhorses for data and network analysis, but also for program analysis and model checking tasks for software verification. A cornerstone problem in directed graphs is the computation of strongly connected components (SCCs). The graphs used in model checking are tremendous, and their explicit representation would not fit in the memory of a computer. Fortunately, one can use symbolic graph representation, in our case Binary Decision Diagrams (BDD) [1].

We recently devised a new symbolic SCC algorithm based on BDDs, that is now the state of the art [2,3]. A single BDD represents a set of nodes in the graph, so operations on the BDD correspond to transforming sets of states. We have a pen-and-paper proof of the correctness of the algorithm, and of the complexity claim that even in the worst case, we only need a linear number of BDD operations.

The challenge of this project is to verify this symbolic SCC algorithm by means of a program verification tool (for instance Dafny [4]) or an interactive theorem prover (for instance Isabelle [5] or Coq [6]). The project can be addressed incrementally through a number of steps:

1. Verify that the symbolic algorithm always provides the correct answers (functional correctness)
2. Verify that the symbolic algorithm needs at most a linear number of symbolic operations (complexity)
3. Generalize the algorithm by making the evaluation order more liberal (towards optimisation)

This project requires interest in computability and logic, graphs and automata.
The main goal is to develop a formal description and derivation of the algorithm.
In particular step 3 lends itself also quite well for implementation and experiments.

Sources:

[1] See Chapter 6:1-3 of Huth & Ryan, “Logic in Computer Science” on BDDs
[2] R. Gentilini, C. Piazza, A. Policriti, Computing strongly connected components in a linear number of symbolic steps
[3] C.A. Larsen, S.M. Schmidt, J. Steensgaard, A.B. Jakobsen, J. van de Pol, A. Pavlogiannis, *A Truly Symbolic Linear-Time Algorithm for SCC Decomposition* (submitted to a conference, available upon request)
[4] Dafny. https://dafny.org/dafny/OnlineTutorial/guide and https://github.com/dafny-lang/dafny
[5] Isabelle theorem prover, https://isabelle.in.tum.de/documentation.html
[6] Coq theorem prover, https://coq.inria.fr/documentation

## Preprocessing Quantified Boolean Formulas for Two-Player Games

With Andreas Pavlogiannis and Jaco van de Pol

Traditional SAT solvers decide the satisfiability of propositional formulas. Although this is an NP-complete problem, progress in the last decade has shown that practical formulas with millions of clauses over thousands of propositional variables can often be solved. As a result, SAT solvers are a popular backend for all kinds of planning problems and model checking problems. However, translating model checking and planning problems to SAT instances leads to very large formulas.

Recently, we considered QBF as an alternative. These are Quantified Boolean Formulas, see [1] for a quick introduction and small examples. The advantage of QBF is that it is much more concise, allowing for efficient translations of planning problems and even two-player games. Also, quantifier alternations can very naturally describe the alternating moves of two players in a game. The disadvantage is that the problem of solving QBF formulas is even harder, it is PSPACE-complete.

We have first encoded classical planning problems (available in a PPDL [2]) into very concise QBF formulas [3]. The generated formulas can be solved by any available QBF solver (like CAQE). More recently, we devised a domain specific language to define two-player board games, like Hex, Connect-Four, Generalized Tic-Tac-Toe, Pursuer-Evader, Breakthrough, etc. We also provide an automated translator to concise QBF formulas. Our experiments show that QBF solvers can solve small and medium game instances, but no large instances of games that require many moves.

The goal of the project is to study preprocessing techniques. These techniques transform QBF formulas into simpler formulas, and are used before calling the final QBF solver. We hope that we can improve current QBF preprocessors, by exploiting the particular patterns introduced by our encoding techniques for planning problems and 2-player games.

The project has a number of goals, in increasing complexity:

a) Study current techniques in QBF preprocessors, for instance Bloqqer [5] and HQSpre [6]
b) Test the effectiveness of QBF preprocessors on a sample of benchmarks (maybe literature study)
c) Test the effectiveness of preprocessors on our encodings of planning problems and 2-player games
d) Improve QBF preprocessors, making use of special patterns in our encodings.
e) For theory-inclined students: study/apply the concept of treewidth in QBF preprocessing [7] [8]

[1] https://en.wikipedia.org/wiki/True_quantified_Boolean_formula
[2] https://en.wikipedia.org/wiki/Planning_Domain_Definition_Language
[3] I. Shaik and Jaco van de Pol, Classical Planning as QBF without Grounding. ICAPS 2022
[4] I. Shaik and Jaco van de Pol, *Two-Player Games as QBF.* To be submitted, available on request.
[5] F.Lonsing, F. Bacchus, A. Biere, U. Egly, M. Seidl: Enhancing Search-Based QBF Solving by Dynamic Blocked Clause Elimination. LPAR 2015. See also the results from the Bloqqer tool.
[6] R. Wimmer, S.Reimer, P. Marin, B. Becker, HQSpre – An Effective Preprocessor for QBF and DQBF
[7] L.Pulina and A.Tacchella, An empirical study of QBF encodings:from treewidth to useful preprocessing
[8] G. Charwat, S. Woltran: Expansion-based QBF Solving on Tree Decompositions (see also DynQBF tool)

## 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.

Currently, IETF standards are only human-readable. The hacspec language (a safe subset of rust) makes them machine readable. In this project, you will write a reference implementation of a number of key cryptographic primitives, 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.

## Refinement types 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.
Each year millions of dollars are lost due to simple programming bugs.
In this project, you will combine flux' refinement types with the rust smart contract language. Refinement types are very expressive types that help to capture errors early.

Flux: https://arxiv.org/abs/2207.04034

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

# 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)