Bachelor Project Proposals 2024

Logic and Semantics & Programming Languages Groups

General information

The projects in this specialization 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 (programming language design, 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 practice 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)

QuickCheck

QuickCheck frameworks are testing tools used in software development to perform property-based testing. Instead of writing specific test cases, developers define properties that their code should satisfy. These properties are then tested against a large number of automatically generated random inputs. QuickCheck frameworks generate a wide range of test inputs to explore various edge cases and potential issues in the code. If a property fails for a particular input, the framework automatically simplifies the input to find the simplest case that causes the failure. This helps developers identify and fix bugs in their code efficiently.

The aim of this project is to (1) explore the design space of QuickCheck frameworks and to (2) design and implement a QuickChecker for the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language.

Package Management

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 fully-featured package manager for the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language.

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 number of unit tests that can be written is limited. Instead, compiler fuzzing techniques have been proposed. A compiler fuzzer 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.

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.

The aim of this project is to: (1) explore the design space of code formatting 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.

Tail Recursion Modulo Cons (co-advised with Amin Timany)

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).

The aim of this project is to: (1) understand tail recursion modulo cons, and to (2) implement it in the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language.

Termination Analysis (co-advised with Amin Timany)

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.

The aim of this project is to: (1) explore algorithms for automatic termination checking, and to (2) implement one of them in the Flix programming language. The work will include reading papers, language design, and implementation in a real-world programming language.

Andreas Pavlogiannis

Predicting memory bugs in concurrent programs

Concurrency is hard to get right, as a programmer must have a thorough mental model of the program that accounts for all the complex ways that different threads interact. Concurrency bugs are also difficult to catch and debug for, as we typically do not have control over the scheduler. You run your program once, and it crashes – you run it a second time, and it works correctly! For this reason, concurrency bugs have been coined as “*Heisenbugs*”.

One technique for analyzing concurrent programs is called *predictive analysis*. We first run the program, and monitor its execution, thereby obtaining a concurrent trace, which is typically non-buggy. Then we analyze the trace, and reason about alternative thread interleavings that could have taken place and that would lead to a bug. This way we *predict* bugs, as opposed to waiting for them to appear by themselves.

This project aims to develop new techniques for predicting memory bugs in concurrent programs. Memory bugs arise when memory accesses are improper. For example, Use-After-Free-type of vulnerabilities occur when one thread tries to read from a file that has been closed by another thread. The task will be to extend some basic predictive techniques to this particular problem, prove them correct, implement them and test them.

Related literature:

[1] https://dl.acm.org/doi/10.1145/3180155.3180225

[2] https://doi.org/10.1145/3468264.3468572

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.
In this area, correctness is very imperative. In this project, you will write safe programs related to Wasm.
For instance:
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 safe compilation, either by using property based testing, refinements or a proof assistant.

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

Parsing

WebAssembly has a text format (.wat), and a binary format (.wasm).
The goal of the project is to implement and verify a parser and printer for the binary format of Wasm, and to connect it to a formalisation of the WebAssembly (which uses the text format).
https://webassembly.github.io/spec/core/binary/index.html

Formalising image formats

Image formats range from the very simple (BMP) to the sophisticated (e.g. JPEG2000).
The goal of the project is to implement decoders and encoders for image formats, and verify that they are inverses (decoding and re-encoding yields the same image).
https://ieeexplore.ieee.org/document/10026566

Formalising PostScript

PostScript is a page description language (like PDF, of which it is the ancestor), a language that describes the appearance of a printed page at a higher level of abstraction than a bitmap would. Structurally, PostScript is a simple stack language with graphic primitives.
The goal of the project is to formalise the semantics of (parts of) PostScript, to implement an interpreter for PostScript that rasterises pages, and to verify it against the specification.
https://archive.org/details/postscriptlangua00adobrich

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

Modeling and Solving Two-Player Games with Quantified Boolean Formulas

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. We now consider QBF as an alternative to specify games. QBF are Quantified Boolean Formulas, see [1] for a quick introduction and small examples. By using quantifiers (for-all and exists), we can model the alternating moves taken in two-player games. The price is: solving QBF formulas is PSPACE-complete.

We have encoded classical planning problems (available in a PPDL, Planning Domain Definition Language [2]) into concise QBF formulas [3]. The generated formulas can be solved by any available QBF solver (like CAQE or DepQBF). More recently, we devised a domain specific language BDDL (Board-Game Domain Definition Language) to define two-player board games, like Hex, Connect-Four, Generalized Tic-Tac-Toe, Pursuer-Evader, Breakthrough, etc. We also provided an automated translator to concise QBF formulas. Our experiments show that QBF solvers can solve small and medium game instances [4].

The goal of this project is to extend the specification language BDDL to define other 2-player games. Take for instance chess: This would require to specify different pieces (BDDL currently considers only black and white stones) and it would require that pieces can move any distance over empty fields (e.g., queens, rooks, bishops). But chess is just an example, we envisage a whole library of game definitions in BDDL.

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

a) Extend the BDDL language and specify more (board) games, for instance chess
b) Develop simple tools to play, solve and visualize any game specified in (extended) BDDL
c) Extend the translation to QBF for the class of extended BDDL games (and use QBF solvers).
d) Experiment with/improve the efficiency of the solver, for instance on chess end games.

[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, Concise QBF Encodings for Games on a Grid. Arxiv 2023

A Protocol for Secure Key Distribution

With Jaco van de Pol (Logic and Semantics) and Diego Aranha (Security)

This problem is defined by one of our industrial partners, in a project on security of Internet-of-Things, using threat modeling and threat analysis.

Problem: key distribution and management on embedded systems is challenging. Most of the devices are resource-constrained and do not have tamper-proof hardware to generate and store keys in a secure manner.

Proposed solution: starting from traditional public-key infrastructures (PKIs), a manufacturer is assumed to hold a key pair able to authenticate long-term keys for all produced devices. These keys, together with the corresponding certificates, are then preloaded in the devices before they are deployed in the field. Using the long-term keys, the devices can run key agreement protocols to establish short-term keys for encrypted communication.

Objectives (depending on the available resources):

(a) Design the protocols and implement the software to bootstrap the proposed solution;
(b) Validate the security properties by simulating simple attack;
(c) Apply modeling and analysis techniques to investigate security (e.g., automata, attack-defense trees)

Applications: the solution could be illustrated within an end-to-end-encrypted machine-to-machine communication framework, where devices are able to authenticate public keys belonging to other nodes and establish TLS connections for secure communication.

Stretch goals: after a prototype of the basic functionality is built, extensions could include periodical key rotation or secure revocation of compromised keys. A follow-up version could also potentially implement a gossiping protocol to avoid the dependence on a centralized authority.

Roles: Industrial partner serves as technical advisor, AU serves as academic advisor.

Scope: 1-3 BSc or MSc students, with scope calibrated with the resources available.

Simplification of Quantum Circuits

With Jaco van de Pol

Quantum Computing [1] promises faster algorithms than traditional computers, but current quantum computers are quite small and noisy [2].

Quantum Algorithms are often specified by Quantum Circuits, which can be quite easily read and manipulated, using IBM's open source tool Qiskit [3].

The goal of this project is:

- To understand the basic working of quantum circuits [1]
- To simplify quantum circuits, for instance by canceling "opposite gates" systematically
- To map quantum circuits to quantum computers, by "swapping" quantum bits to physical neighbors [4]

Resources:

[1] Wikipedia on Quantum Computing, Quantum Circuits, and Quantum Gates
[2] John Preskill, Quantum Computing in the NISQ era and beyond, 2018
[3] IBM Qiskit (tools for simulation and manipulation of quantum circuits, and documentation)
[4] Our paper "Optimal Layout Synthesis for Quantum Circuits as Classical Planning", ICCAD 2023

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, rust, ...

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/ ("ocaml/F* with refinement types")
https://www.manning.com/books/type-driven-development-with-idris
http://adam.chlipala.net/cpdt/
https://ucsd-progsys.github.io/liquidhaskell/
Rust (flux or hax refinement types)

High assurance cryptography

With Bas Spitters and Diego Aranha

High Assurance Cryptography (https://hacspec.org/)

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 hax' refinement types with the rust smart contract language. Refinement types are very expressive types that help to capture errors early.

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)