Bachelor Project Proposals 2026

Programming Languages, Logic, and Software Security

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:

- Introduction to Programming (Magnus Madsen)
- Programming Café (Magnus Madsen)
- Computability and Logic (Jaco van de Pol, Daniel Gratzer)
- 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 Programming Languages, Logic, and Software Security, 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)
- Daniel Gratzer (proof assistants, type theory, denotational semantics)
- 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 (automata, quantum circuits, model checking, games)
- 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

Troupe

Troupe is a research programming language for secure distributed and concurrent programming. A characteristic feature of Troupe is its runtime security monitoring that enables Troupe programs to execute untrusted code securely. Troupe's compiler is written in Haskell, and the runtime is written in Typescript.

Under the Troupe umbrella, there is a possibility for many projects, both in security and privacy (that primarily focus on Troupe applications) and programming languages (that focus on extending the Troupe compiler/runtime). Here's a list of a few example projects. You are welcome to approach me with your own ideas.

Privacy Aware User Interface Library in Troupe, with applications to Signal

Personal computing devices handle our sensitive information, but it is rare for a personal computing device to be truly personal. In many professional and social situations, what others can learn by looking at our screens may have problematic consequences. In this project, you will work on the design and implementation of a privacy-aware user interface library. The general idea is that programs should react to changes in the operating environment, such as plugging the computer into a projector, starting screen sharing, or changing location, for example, while on public transportation.

Concretely, this project will require developing a UI library in Troupe (there is currently none), either as a modern CLI app or a browser-based library with DOM interaction. Reacting to privacy-changing events should not be unlike how traditional UI libraries respond to other changes. For instance, similar to "onWindowResize" in regular UI programming, we can imagine an event like "onPrivacyContextChange(newContext)" or something similar. The idea is to rely on Troupe's security monitoring to guarantee that the UI does not leak information. As a prototype to demonstrate the utility of the library, you may develop a couple of prototypes, e.g., a small task management app where task entries have different confidentiality levels, and more interestingly, a minimal Signal Instant Messaging client that is privacy-aware, e.g., with different security classifications of contacts.

Secure agentic coding

Troupe's design predates the modern LLMs-based agentic coding, but the underlying threat model is a good fit for handling untrusted code authored by LLMs. While the original threat model has been designed to handle incompetent/malicious programmers, it happens to be just the right approach when we consider incompetent/malicious AI. The idea is to leverage LLMs for Troupe code generation at scale, but mitigate the security risks using Troupe's security mechanisms. Our preliminary experiments suggest that modern LLMs can already generate Troupe programs (with access to the language reference manual and test corpus), but further work is required to investigate both limitations and potential. We expect the work to focus on incorporating security concepts used in Troupe, such as security labels and capabilities (that lack in conventional programming and hence in the training sets), into the LLM-based workflow, and prototype

applications that leverage LLM-based code generation. One concrete example of such an application is a grading system for a programming course where the grading policy is specified in natural language, but the LLM-generated assessment code must not leak information when analyzing the test outcomes.

Efficient serialization

This project will focus on extending the Troupe backend to use Google Protobufs for efficient serialization. The project will involve working on both the compiler and the runtime, setting up distributed systems for empirical measurements, extending the language runtime to induce controlled failures for stress testing, and developing a corpus of benchmarks.

Self-hosting IR compiler

Troupe frontend in Haskell consists of two phases: surface-level syntax to IR and IR to JavaScript. The second phase is particularly interesting because IR is considered untrusted, and this part implements security monitoring and a series of optimizations. Rewriting the backend in Troupe itself will improve the potential for porting the language to other platforms, i.e., eliminating the need to call a Haskell application (either as a binary or a hosted service).

Diverse backends

The only maintained backend we have for Troupe right now is a Node.js CLI application. In this project, you will work on extending the runtime to enable Troupe to run on other backends, such as browsers, iOS, and Android.

Further reading:

- Troupe website at AU: http://troupe.cs.au.dk

Lars Birkedal (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

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

Magnus Madsen (magnusm@cs.au.dk)

The Flix Programming Language

Flix (https://flix.dev/) is an effect-oriented programming language under development at Aarhus University.

Flix combines functional, imperative, and logic programming with a sophisticated type and effect system that supports effect handlers and region-scoped mutable memory. Flix also features an advanced trait system with support for higher-kinded types and associated types and effects.

If you are interested in advanced programming languages, type and effect systems, or compilers, we have several exciting projects:

- **(Code Formatter)** Integrate a built-in code formatter into the Flix compiler.
- **(Program Optimizations)** Introduce new program optimizations, such as tail recursion modulo cons, unit and singleton elimination, and unboxed options.
- **(Faster Type Inference)** Improve the performance of type and effect inference through advanced data structures, such as Zhegalkin polynomials.
- **(Improved Effect Errors)** Extend the Flix type and effect system with effect provenance to better capture the introduction and elimination of effects with a view towards improved effect error messages.
- **(Termination Checker)** Add a simple termination checker to the Flix compiler based on the Foetus algorithm used in Agda.
- **(Semantic Grep)** Add support for semantic grep to enable compiler-powered queries over the structure of Flix programs. Useful for AI agents.
- **(Reworked CodeGen)** Rework the Flix backend to generate JVM bytecode using the new ClassFile API, and explore alternative encodings of effect handlers.
- **(Library Extensions)** Extend the Flix Standard Library with new algebraic effects and default handlers.

Additional Resources

- flix.dev
- doc.flix.dev

Jean Pichon

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

Note: There is scope for other Sail-based projects.

Mechanised properties of image formats

Lossless image formats range from the very simple (for example BMP) to the sophisticated (for example PNG and lossless JPEG).
The goal of this project is to implement image formats (like BMP or PNG) in the form of their encoding and decoding functions, and of showing that these two functions are mutual inverses: decoding and re-encoding yields the same image.

Formally Verified Quite OK Image Format, Kunčar
https://ieeexplore.ieee.org/document/10026566

Mechanised definition of the Postscript language

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 stack language whose semantics is, unusually, an image.
The goal of this project is to mechanise the definition of (a significant subset of) the Postscript language and of its operational semantics, and (or: in the form of) an interpreter for PostScript that rasterises pages.

Postscript Blue Book: “PostScript Language Tutorial & Cookbook” https://archive.org/details/PSBlueBook
Postscript Green Book: “PostScript Language Program Design”
https://archive.org/details/PSGreenBook/page/n7/mode/2up

Safe compilation to LLVM IR and WebAssembly (co-advisor Bas Spitters)

WebAssembly is a language designed to be the compilation target to run programs in web browsers, but it is also gaining traction for edge computing and blockchains. 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 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.
Both Wasm and LLVM can be targeted from functional languages too. Here, lambda-ANF is often used as an intermediate language. Exploring how much could be shared by two such backends would be interesting.
In short, 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

Jaco van de Pol

Simplification of Quantum Circuits

(joint with Irfansha Shaik @ Kvantify)

Quantum computing [1] promises fundamentally faster algorithms than traditional computers. Quantum algorithms are often specified by quantum circuits, which can be quite easily read and manipulated using IBM's open source python package Qiskit [2]. But current quantum computers are quite small and noisy [3], limiting what you can do in practice. This motivates rigorous simplification of quantum circuits, thus enabling larger experiments.
We propose to use classical planning techniques [4], SAT solving [5], and graph algorithms to do so. Our current research is codified in the Q-Synth tool [7] (see tutorial). This tool allows us to optimize [5] and map [4] quantum circuits to quantum computers.

There are several possible subprojects, ranging from theory, development, experiments:

- Further integration of circuit optimization and circuit mapping [4,5]
- Extension of quantum circuit optimization to other quantum gate sets [5,6].
- Optimized Compilation of higher-level quantum programs to circuits.
- More efficient planning and/or SAT encodings and improved solver strategies
- Extensive experimentation with encodings / solver tools / high-performance hardware
- Application of quantum circuit optimization to quantum algorithms and quantum error correction

Resources:

[1] Wikipedia on Quantum Computing, Quantum Circuits, and Quantum Gates
[2] IBM Qiskit (tools for simulation and manipulation of quantum circuits, and documentation)
[3] John Preskill, Quantum Computing in the NISQ era and beyond, 2018
[4] Our paper "Optimal Layout Synthesis for Quantum Circuits as Classical Planning", ICCAD 2023
[5] Our paper "Optimal Layout-Aware CNOT Circuit Synthesis with Qubit Permutation", ECAI 2024
[6] Our paper "CNOT-Optimal Clifford Synthesis as SAT", SAT 2025
[7] Q-Synth tool (python) and jupyter notebook tutorial

Quantum Programming Languages

(joint Jaco van de Pol & Bas Spitters)

Quantum computing [1] promises fundamentally faster algorithms than traditional computers. Quantum algorithms are often specified by quantum circuits, which can be quite easily read and manipulated using IBM's open source python package Qiskit [2]. But quantum circuits are pretty low-level to work with, and they also lack certain modularity and control mechanisms (such as while-loops).

In this project, we will study some proposals for high-level quantum programming languages [3,4,5], and depending on your taste look into:

- Implementing known quantum algorithms in a high-level quantum programming language
- Writing a simple interpreter to simulate programs in a quantum programming language
- Compile a (simplified) quantum programming language to quantum circuits

[1] Wikipedia on Quantum Computing, Quantum Circuits, and Quantum Gates

[2] IBM Qiskit (tools for simulation and manipulation of quantum circuits, and documentation)
[3] QRISP (Fraunhofer Institute)
[4] Guppy (Quantinuum)
[5] Q# (Microsoft) -

I/O-efficient Binary Decision Diagrams

(co-adivsor Steffan Sølvsten)

One way to verify the correctness of a system is via *Model Checking*. This essentially boils down to checking the correctness of every execution path by mere brute force. But, one quickly runs into the *state space explosion* problem: even moderate systems can have more distinct states than the number of particles in the universe. One way to compress the state space is by use of the *Binary Decision Diagram* (*BDD*) data structure. Yet, even these BDDs can grow larger than the RAM of the machine. At this scale, one has to also optimize for the number of reads from and writes to the disk (I/Os).

In this project, you will extend and/or improve upon *Adiar*, a novel implementation of BDDs that optimises its performance (in theory and in practice) not only for computation time but also space usage and number of I/Os. Depending on your interest in algorithmic engineering and/or application of BDDs, this project can set out to:

- Improve performance of the currently implemented BDD operations.
- Implement missing BDD operations.
- Add support for other types of decision diagrams.
- Extend the accompanying benchmarking suite with other applications of BDDs.

Resources

1. Steffan Sølvsten and Jaco van de Pol. Adiar: Binary Decision Diagrams in External Memory, TACAS 2022
2. Adiar's source code. github.com/ssoelvsten/adiar, GitHub
3. Benchmarking Suite source code. github.com/ssoelvsten/bdd-benchmark/, GitHub

Bas Spitters

spitters@cs.au.dk

High assurance cryptography
(co-advised by Diego Aranha)

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 (https://hacspec.org/) 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.

The current post-quantum transition is often a twin transition to **high-assurance** PQ software. See for example the Signal messenger, or iMessage. Post-quantum implementations would be a good project topic.

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

Security by Design using LLMs for code

Large Language models are used more and more for generating code. However, they are known to produce unsafe code. There are a number of suggested solutions, most of them involving formal methods/symbolic AI. E.g. involving a type checker, a testing framework, an automatic/interactive theorem prover, …
In this project, we will explore a combination of these techniques:

- Automatic synthesis of invariants, unit/property based tests, assertions, and other proof structures.
- Code repair
- Automatic translation between languages
- …

The project will focus on highly structured languages such as ocaml, rust, …
The project will consist of a combination of literature exploration, implementation and experimentation.

This is a follow-up of a project together with Alexandra Institutet, who might also be involved in this project.

Amin Timany (timany@cs.au.dk)

Relational Reasoning

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.

References

- 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)

Amin Timany (timany@cs.au.dk)

Implementation of an authoritative DNS server

Most data processed by computers are neither human-friendly nor stable, *e.g.*, IP addresses. Domain Name System (DNS) provides a way to tie data used by computers to a *domain name*, which is simple for humans to read and remember. For example, when browsing 'au.dk', computers connect to the server whose IP address is 185.45.20.48[1]. DNS provides a mapping between names, and the data used by computers, *e.g.*, IP addresses, what server to deliver emails to, what fallback server to use if another server is down. DNS consists of two main components: **authoritative DNS servers** and **DNS resolvers**. The former provides aforementioned mappings. The latter queries authoritative servers to obtain those mappings.

DNS is a structured distributed system. Names are a list of dot-separated strings, *e.g.*, '[au; dk]', allowing partitioning the mapping in so-called zones. Zones form a tree-like structure. In our example, three zones are in use: '.' (the root-zone), 'dk.' and 'au.dk.'. The tree-structure can be observed, *e.g.*, at https://dnsviz.net/d/au.dk/dnssec/. This tree-structure and decentralization are reflected when making a DNS query. For example, when one wants to obtain the IP address corresponding to 'au.dk', they use a DNS resolver that will perform the following steps.

1. Ask where 'au.dk.' is located to servers managing the '.' zone. The response includes a list of servers handling the '.dk.' zone. This is known as delegation. That is, the '.' zone delegates the query for '.dk.' to these servers.
2. The resolver picks one of these servers, *e.g.*, 'b.nic.dk'. This server returns a list of two servers managing the 'au.dk.' zone.
3. The resolver picks one of these servers which then provides the resolver with the IP address of 'au.dk'.

In order to increase trust in DNS, responses to queries can be cryptographically signed by authoritative DNS servers. The protocol for signed DNS responses is called DNSSEC. In order to verify a response one gets from a server, one can check the signature of the response using the key of the zone. To ensure that a zone's keys are legitimate, parent zones use signed/verified delegations. The blue arrows on the aforementioned Web page https://dnsviz.net/d/au.dk/dnssec/ represent signed delegations. They illustrate how one can trust the zone 'au.dk' (provided they trust the key of the root zone '.'). Hence, responses of 'au.dk' can be verified. However, as of this writing, answers of 'ku.dk' cannot be verified, as illustrated on https://dnsviz.net/d/ku.dk/dnssec/.

Finally, in order to make DNS more resilient, all zones are replicated on at least two different name servers, *e.g.*, 'ns3.au.dk.' and 'ns4.au.dk.' in the case of the 'au.dk.' zone. Any zone update on one server must be forwarded to other servers.

The aim of this project is for students to implement an authoritative DNS server in OCaml. The DNS server should serve one or more DNS zones and allow administrators to update zones at runtime. Moreover, the implementation should support DNSSEC and replication. Support for DNSSEC means that every zone served by the server can be signed. Moreover, should there be no data to answer to a query, the server should provide a proof of non-existence (e.g. through NSEC3 responses). Replication can take several forms. We suggest using a leader/follower model, *i.e.*, for each zone, choose one authoritative server as reference. Updates should be performed on this server, and then propagated to its replicas.

[1] Or 10.83.252.23 when connected to the university network.

References:

- Domain names concepts: https://www.rfc-editor.org/rfc/rfc1034.html
- DNSSEC specification https://www.rfc-editor.org/rfc/rfc9364
- Zone transfer-relevant RFCs:
    - https://datatracker.ietf.org/doc/html/rfc5936
    - https://datatracker.ietf.org/doc/html/rfc1995
    - https://datatracker.ietf.org/doc/html/rfc1996

  Note: Incremental zone transfer (the last two RFCs) is optional for this project.

Amin Timany (timany@cs.au.dk)

Implementation of a subset of the QUIC protocol

The Internet Protocol (IP) enables sending datagrams from one machine to a set of other machines on the same IP network. On top of this protocol run the so-called transport protocols, which enable communication between software running on different machines.

The two main transport protocols used today are TCP and UDP. TCP establishes a reliable connection between two end-points. If this connection is broken, it is lost, and needs to be re-established by applications. TCP is used by applications that need reliable communication between end-points. UDP on the other hand allows an application to send datagrams to another application on the same IP network. Unlike TCP, UDP does not provide any guarantee of delivery: data can be lost, delivered out-of-order, or duplicated. Thus, UDP is mostly used by applications that do not need reliable communication, *e.g.*, DNS servers.

QUIC is proposed as a lightweight alternative to TCP. Similarly to TCP, QUIC provides reliable communication between two applications. However, unlike TCP, QUIC is more resilient to network issues. That is, the connection can survive even if the network is temporarily unavailable. Moreover, QUIC provides more features than TCP alone. It notably encrypts data between clients and servers using TLS (Transport-Layer Security protocol), and enables multiplexing, *i.e.*, several streams of data can be sent and received through a single connection.

The core idea of QUIC is to implement reliable communication channels on top of UDP. The aim of this project is to write a QUIC implementation in OCaml. QUIC is an extensive protocol and provides numerous features. Most of them do not need to be implemented in this project. The aim is to implement a library with at least the minimal features enough to support writing simple QUIC servers and clients.

References

- QUIC-related RFCs:
  - https://datatracker.ietf.org/doc/html/rfc8999
  - https://datatracker.ietf.org/doc/html/rfc9000
  - https://datatracker.ietf.org/doc/html/rfc9001
- UDP RFC:
  - https://datatracker.ietf.org/doc/html/rfc768

Daniel Gratzer (gratzer@cs.au.dk)

Design and implementation of type systems

Many modern programming languages feature advanced static type systems designed to prevent programmer error prior to a program's execution. This project will focus on understanding the fundamentals of these static type systems and gaining hands-on experience with their implementation. Concretely, it will consist of

1. Understanding how static type systems are specified mathematically (judgments, inference rules, etc.)
2. Actually implementing type-checkers for one or more type systems in order to see how these rules relate to actual implementation.

The precise type systems to be considered are negotiable. The minimal version of this project would focus on the simply-typed lambda calculus (a purely functional language with higher-order functions) and several extensions thereof (adding records, algebraic data types, and recursion). More advanced versions of the project might consider polymorphic type systems such as System F---note, we do not consider type inference as in Lars Birkedal's project---or even simple dependently typed systems.

There is quite a lot of flexibility possible for the precise type systems and, especially, implementation language that students may use. The final product, however, must consist of an implementation of one or more static type systems along with a written report which includes details on their formal specification.

Standard references for type systems (including details on implementation) may be found in

[1] *Types and Programming Languages.* Benjamin C. Pierce
[2] *Practical Foundations of Programming Languages.* Robert Harper

Daniel Gratzer (gratzer@cs.au.dk)

Compiling higher-order programming languages

In the compilers course, we saw how to compile high-level features like functions, loops, and into a lower-level language (LLVM bytecode). This project focuses on compiling away one particular feature: higher-order functions. In particular, it explores taking a minimal language, a variant of scheme, with higher-order functions and essentially nothing else and compiling it down to a lower-level language.

As we shall see, compiling away higher-order functions can be quite an involved process. We shall follow a multi-step process which includes closure conversion, continuation-passing transformation, and finally code generation. This project builds on top of the compilers course and expands its reach into more functional languages. While the core of the project is fixed, there are myriad opportunities for extension if time permits. For example, for an especially motivated student, it may be possible to extend their Dolphin compiler with higher-order functions.

The aims of this project are to (1) learn about the implementation issues associated with higher-order functions and (2) develop a practical understanding of how these issues can be resolved.

[1] *Modern Compiler Implementation in ML.* Andrew W. Appel (Chapter 15)
[2] *Essentials of Compilation.* Jeremy Siek (Chapter 8)

Daniel Gratzer (gratzer@cs.au.dk)

Verified functional programming with Agda

This project focuses on using a dependently typed programming language (Agda) to program various simple algorithms which are *provably correct.* That is, rather than writing an algorithm and extensively testing it or making a pen-and-paper argument for its correctness, we will use Agda to define a version of e.g., merge sort which is known to be correct the moment it compiles.

This project will then involve learning the foundations of dependently-typed programming using one of several books on Agda. The focus will be on obtaining a "practical" (as opposed to theoretical) understanding of Agda. Once we have established some basic familiarity with Agda, we will choose from a number of small algorithms and define one in Agda along with a proof of its correctness. Examples of possible algorithms include:

1. Merge sort
2. Euclid's algorithm for the greatest common divisor
3. A simple regular expression engine
4. A naive prime factorization algorithm
5. ....

While this project focuses on using Agda, rather than studying its theory, comfort with basic discrete math and proofs will be essential. Experience with functional programming will also be very useful.

[1] *Verified Functional Programming in Agda.* Aaron Stump
[2] *Programming Language Foundations in Agda.* Philip Wadler, Wen Kokke, Jeremy G. Siek