MASTER THESIS PROJECT PROPOSAL

Towards a System-Level Functional Language

Sebastian Selander
gusselase@student.gu.se

Samuel Hammersberg
gushamsak@student.gu.se

Supervisor at CSE: Jean-Philippe Bernardy

Relevant completed courses:

Sebastian Selander:
DIT235, Types for Programs and Proofs
DIT260, Advanced Functional Prog
DIT261, Parallel Functional Prog
DIT203, Logic in Computer Science
DIT301, Compiler Construction

Samuel Hammersberg:
DIT235, Types for Programs and Proofs
DIT260, Advanced Functional Prog
DIT261, Parallel Functional Prog
DIT301, Compiler Construction

December 13, 2024

1. Background

System-level programming is the act of developing software that interacts directly with a computer's hardware, or providing foundational services to other software. We will define system-level programming as Wikipedia [1] defines it:

- Programs can operate in resource-constrained environments
- Programs can be efficient with little runtime overhead, possibly having either a small runtime library or none at all
- Programs may use direct and "raw" control over memory access and control flow
- The programmer may write parts of the program directly in assembly language

For system-level programming today developers mostly choose between procedural programming languages, for instance: C, C++, or Rust. In C memory management is manual, and in C++ there are options for automatic memory management, but the compile time guarantees are few and weak. Rust has mostly solved the issue of memory management with the introduction of an ownership model and a borrow checker [2].

Although Rust offers many functional aspects, many are still missing, for instance; *purity* and *referential transparency*. We propose an alternative that addresses these points, the programming language: *System-level Functional Language* (SLFL).

1.1. Motivation

Functional programming tends to emphasize *referential transparency*, *higher-order functions*, *algebraic type systems*, and *strong type systems*. Although the merits of functional programming are evident [3], it is under represented for system-level programming. The reason functional languages are not used in system-level programming is the lack of predictable performance. Unpredictable performance can be traced back the use of *immutable* data structures, as opposed to *mutable* data structures. The former requires copying, and subsequently a form of automatic memory management, at least for convenience, whereas the latter can be modified in place.

Girard's linear logic [4] is a refinement of classical and intuitionistic logic, where, rather than propositions being truth statements, they represent *resources*, meaning propositions are objects that can be modified into other objects. Linear logic models the problems of shared and mutable data, both

of which are of critical importance in system-level programming. In linear logic, the uses of weakening and contraction are carefully controlled, which in a programming language setting means variables must be used exactly once.

The plan of our thesis is to compile and extend System-Level Functional Language. The language is created by Jean-Philippe Bernardy, which in turn is based on Girard's linear logic. SLFL is proposed as an alternative to system-level languages, as well as being an intermediate compilation target for functional languages.

1.2. Related work

Lilac: a functional programming language based on linear logic [5], gives an implementation of a type inference algorithm, similar in style to Milner's algorithm W [6]. Lilac also extends the lambda calculus with recursion and datatypes. However, the focus of Lilac is not how to compile efficient machine code.

Secondly, in *Efficient Implementation of a Linear Logic Programming Language* [7] the authors give an implementation of a compiler for the linear logic language Lolli. Although they provide a efficient compilation for the language, it is a logic programming language, which SLFL is not.

The paper *A Type System for Bounded Space and Functional In-Place Update* [8] presents a way to compile a linearly typed first-order functional language into `malloc()`-free `C` code. Again, however, this is not satisfactory as we want SLFL to be a higher-order language.

In the master's thesis "Towards a practical execution model for functional languages with linear types" [9] Nordmark implements a virtual machine for a linearly typed language. Unfortunately the virtual machine language is untyped, and because SLFL is linearly typed, it can not be utilized as an intermediate language for compiling to SLFL. As the goal is to create efficient machine code, the virtual machine can not be a target language either.

The goal of our thesis will be to complete the design of SLFL and create a compiler for the language. The language specification will be extended as well, and those extensions will be compiled.

2. Goals and planning

In this chapter we will go into more detail about the extensions to SLFL. We will clarify what the extensions are and explain why they are wanted.

2.1. Compilation

The first goal of the project is to compile SLFL to native machine code. We plan on doing this by compiling to LLVM [10], which in turn can be compiled to native machine code. LLVM provides a simple and efficient abstraction layer, compared to the native machine code alternatives. The Rust compiler and Clang, one of the C++ compilers, both compile to LLVM, demonstrating that LLVM is a reliable option.

2.2. Language extensions

SLFL is currently incomplete, and this chapter will cover some of the extensions that we consider extending the language with. SLFL will consist of several intermedate compilation steps, each linearly typed. As such, for every extension we make to the language, we must also create the corresponding typing rules.

2.2.1. Exponentials

As briefly mentioned in Section 1.1, variables in a linearly typed programming language must be used exactly once. There is an exception to this rule, called exponentials. Exponentials can be used to duplicate or drop values. We currently consider two implementations:

- Implement exponentials using reference counting. This would then require recursive deallocation of any other exponentials used in the value.
- Limit exponentials to non-linear closures, where every value allocated in the closure is allocated using reference counting.

If neither of these solutions work out, time will be spent on creating an alternative, as exponentials are important for flexibility when writing code.

2.2.2. Recursive and contiguous data types

When representing more complicated data types such as different types of trees or list in functional programming, they are almost always implemented using recursive data types. In theory this is fine, but in practice it leads to overhead, such as pointer indirection and poor cache locality. As such, we will investigate adding statically or dynamically sized contiguous data structures, such as arrays or vectors.

2.2.3. Laziness

Laziness, or commonly referred to as call-by-name, means that values and expressions are evaluated only when they are used. Lazy evaluation is an effective tool for achieving modularisation [3].

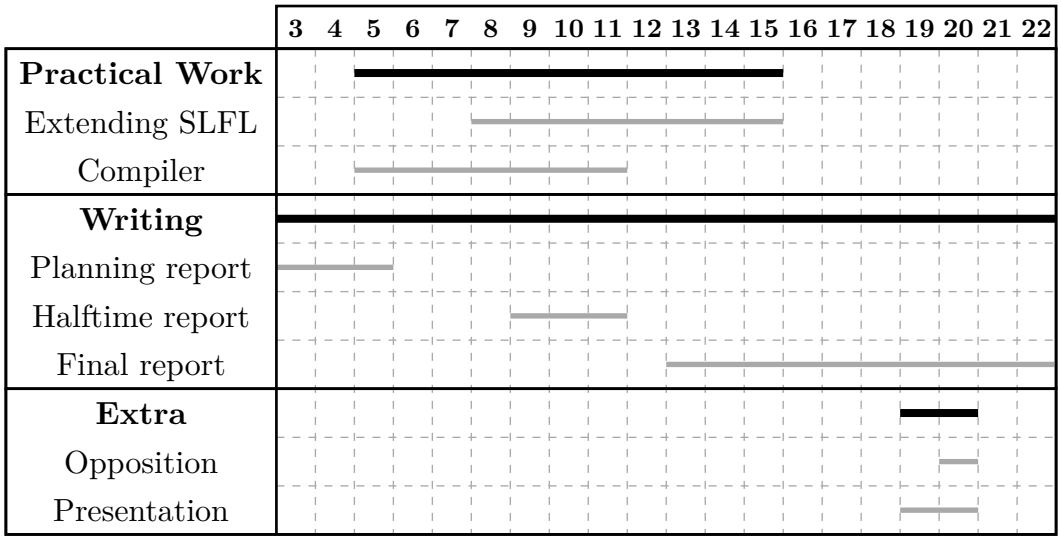
We will explore if lazy evaluation is a feasible addition for a linearly typed system-level language, as well as investigate how this will interact with I/O, where the order of execution is important.

3. Approach

The work can be divided into two parts:
1. Creating a compiler for SLFL as it exists currently.
2. Extending SLFL and the compiler with the new features.

We plan on first focusing on the actual creation of the compiler. Once the base compiler has been made, we will focus on adding the new features to the language.

We have made a preliminary plan of how we will schedule our time:



3.1. Evaluation

Although this is a system-level language, and performance and resource usage is key, the language will still be in the infant stage of development after the thesis. Rather than compare the language to mature pre-existing system-level languages, we will focus on two key aspects:

- Self-evident: Every intermediate compilation step should be typed

- Correctness: Every intermediate compilation step should be type checked.

As SLFL is a system-level language, it is very important that no memory is leaked during the running time of the program. We will measure this using Valgrind [11]. If time allows we will also measure the memory usage of the language most likely using the `glibc` tool `memusage` [12].

References

[1] Wikipedia, "Systems programming — Wikipedia, The Free Encyclopedia." 2024.

[2] N. D. Matsakis and F. S. Klock, "The Rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[3] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.

[4] J.-Y. Girard, "Linear logic," *Theoretical computer science*, vol. 50, no. 1, pp. 1–101, 1987.

[5] I. Mackie, "Lilac: A functional programming language based on linear logic," *Journal of Functional Programming*, vol. 4, no. 4, pp. 395–433, 1994.

[6] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[7] J. S. Hodas, K. M. Watkins, N. Tamura, and K.-S. Kang, "Efficient Implementation of a Linear Logic Programming Language.," in *IJCSLP*, 1998, pp. 145–159.

[8] M. Hofmann, "A type system for bounded space and functional in-place update," in *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 9*, 2000, pp. 165–179.

[9] F. Nordmark, "Towards a Practical Execution Model for Functional Languages with Linear Types," 2024.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[11] "Valgrind Home — valgrind.org."

[12] "The GNU C Library — sourceware.org."