MASTER THESIS PROJECT PROPOSAL

Towards a System-Level Functional Language

Sebastian Selander
`gusselase@student.gu.se`

Samuel Hammersberg
`gushamsak@student.gu.se`

Suggested supervisor at CSE: Jean-Philippe Bernardy

Relevant completed courses:

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

December 13, 2024

Contents

remove the outline

1. Background

1.1. System-level programming

System-level programming is the act of developing software that interacts directly with a computer's hardware, or providing foundational services to other software. The following are some properties of system-level programming as defined on Wikipedia:

- 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

1.2. Motivation (why is SLFL interesting)

There are advantages to functional programming [1]. 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, it is under represented in system-level programming. The reason functional languages are not used in system-level programming is the lack of predictable performance. This lack of predictable performance is frequently traced back to 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 [2] is a refinement of classical and intuistionistic logic, where, rather than proposition 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, as well as alleviating the need of a garbage collector.

System-Level Functional Language (SLFL) is a language based on Girard's linear logic created by Jean-Philippe Bernardy. SLFL is proposed as an alternative to system-level languages, as well as being an intermediate compilation target for higher-level functional languages. The main purpose of this thesis would be to create a compiler for SLFL, as well as extending the language adding useful features, while making sure that those features work in the context of linear types.

Should PLL be referenced here too?

1.3. Pre-existing work in the literature

TODO [citations]

1.4. Pre-existing work in the group

- previous work by nordmark
- stub design/compiler by supervisor

1.5. Transition paragraph

- What is in the scope for this thesis, more precisely

2. Goals and planning

The goal of this thesis is to extend the language of SLFL, as well as building a compiler for it. For the compilation part we preliminary want to compile the language to LLVM [3]. Time will be spent on investigating if LLVM is a good compilation target, and if not, other low level targets such as x86-64 assembly will be considered.

2.1. Language extensions

Currently the language is somewhat simple, and the following sections cover extensions to the language we want to add if time allows.

2.1.1. Exponentials

figure out what this means, closures??

2.1.2. Records

While simple data types suffice in a lot of places, records provide important context to data types, allowing for labeled fields.

2.1.3. 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, bad cache locality and more. Due to this we will also investigate adding statically or dynamically sized contiguous types, such as arrays or vectors.

2.1.4. Laziness

lite osäker på hur vi ska justifya de

2.2. Evaluation

As this is a system-level language, performance and resource usage is key. To evalute how performant the language is, simple programs will be written in SLFL and other system-level languages such as C to compare how well they perform against each other. Primarily execution time and memory usage will be measured, most likely using the GNU tool `time` [4] and the `glibc` tool `memusage` [5] to do so. The SLFL programs will also be evaluated using Valgrind [6] to make sure that memory leaks do not occur.

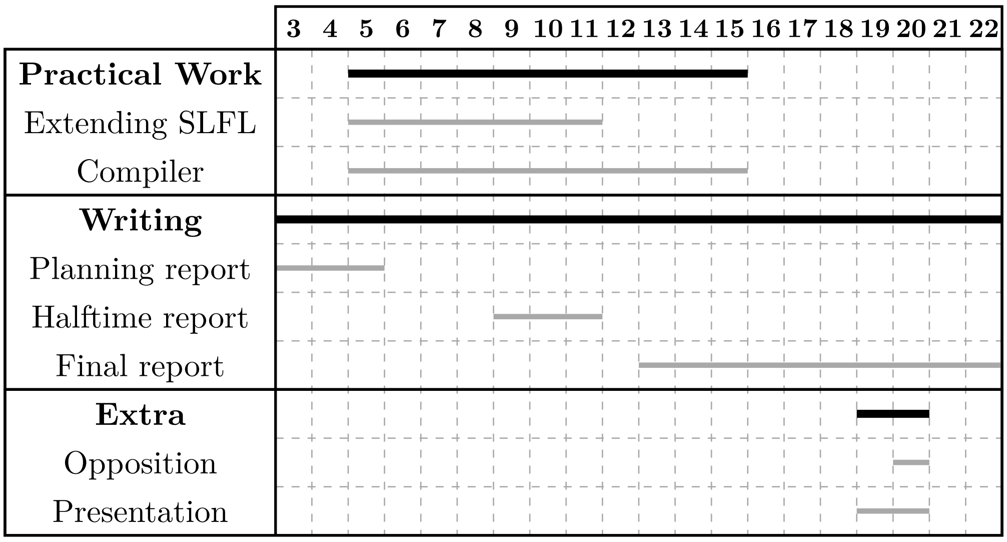

Figure 1: An outline of our planned work schedule

- how it differs from previous work (Nordmark)
  - Nordmark has a special purpose VM.
  - using Nordmark's VM as an intermediate language goes against Principle 1.
- what to do?
  - Translate each construction/typing rule into code
  - if a construction cannot be translated this way: refine it (design a more low-level rule which serves in an intermediate compilation step.)

References

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

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

[3] 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.

[4] “GNU Time - GNU Project - Free Software Foundation — gnu.org.”

[5] “The GNU C Library — sourceware.org.”

[6] “Valgrind Home — valgrind.org.”