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:
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

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)

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 [1], 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 [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.

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.

Should PLL be referenced here too?

correct?

1.3. Pre-existing work in the literature

Introduce the pre-existing work

- Lilac: a functional programming language based on linear logic [3]

- Linear Haskell: practical linearity in a higher-order polymorphic language [4]
- Efficient Implementation of a Linear Logic Programming Language [5]
- A type system for bounded space and functional in-place update [6]
- Efficient Functional Program ming using Linear Types: The Array Fragment [7]

1.4. Pre-existing work in the group

SLFL is a language designed by Jean-Philippe Bernardy with contributions from Filip Nordmark. Bernardy's "vision" is a strongly typed common machine-level language that can be used as an intermediate target.

JP definitely needs to review this

In the master's thesis "Towards a practical execution model for functional languages with linear types" [8] 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.

1.5. Transition paragraph

The scope of the thesis includes creating a compiler for SLFL that utilizes the linearity of the language, possibly extend SLFL with recursive data types, records, and laziness. Unless typing rules for the aforementioned extensions already exist, those will have to be formulated as well.

2. Goals and planning

This chapter will define what the goals are, why they are interesting, and give an estimate of what needs to be done.

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. As SLFL follows a formal specification, all of these new rules have to do so as well. For each new feature new typing rules and new reduction rules will be introduced for working with them.

2.1.1. Exponentials

Add text here!

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

Add text here!

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` [9] and the `glibc` tool `memusage` [10] to do so. The SLFL programs will also be evaluated using Valgrind [11] to make sure that memory leaks do not occur.

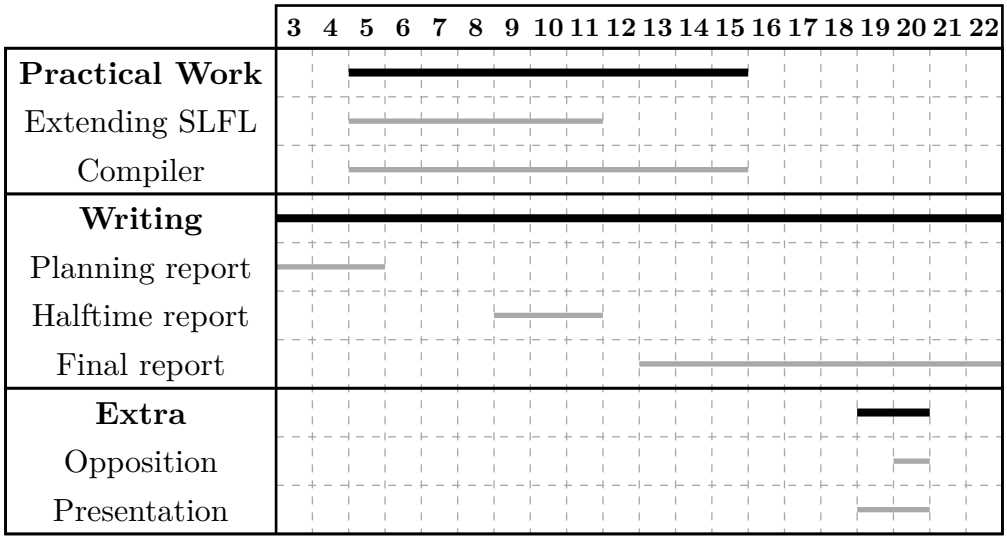

Figure 1: An outline of our planned work schedule

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

[4] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear Haskell: practical linearity in a higher-order polymorphic language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[5] 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.

[6] 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.

[7] V. L. Juan, "Efficient Functional Programming using Linear Types: The Array Fragment," 2015.

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

[9] "GNU Time - GNU Project - Free Software Foundation — gnu.org."

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

[11] "Valgrind Home — valgrind.org."