

System-level Functional Programming with Linear Types

Sebastian Selander and Samuel Hammersberg

System level programming today

In today's world we are blessed with a lot of choices for system level programming:

- C
- C++
- Rust
- And more . . .

System level programming today

Although these languages are great, they are missing some things that some developers enjoy:

- Referential transparency
- Purity
- Strongly typed

Functional programming to the rescue!

What does FP not lack in?

- Referential transparency
- Purity
- Strongly typed

You could of course create a FP language without these

Functional programming to the rescue!

Is system-level programming we often have to care about memory, but most functional languages use garbage collectors!

Due to how most languages are made a lot of copying is done when updating values, and the old value has to be garbage collected

A garbage collector is not preferred for system-level programming as we want control over memory!

We want to know when memory is collected, and what gets collected

Functional programming to the rescue!

So are we stuck with a garbage collector?

Functional programming to the rescue!

So are we stuck with a garbage collector?

No! We can use linear types!

Linear Types

Every variable must be used **exactly once**

- Linear arrow: \multimap
- Normal arrow: \rightarrow

```
const :: a -o b -o c
const a b = a -- error
```

```
append :: [a] -o [a] -o [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys -- good
```

- Some programs no longer compile
- No need to copy the data structure, instead mutate!

Linear Types

What is **using exactly once**? For a function f with arguments x :

Linear Types

What is **using exactly once**? For a function f with arguments x :

- Returning x unmodified.
 - $f\ x = x$

Linear Types

What is **using exactly once**? For a function f with arguments x :

- Returning x unmodified.
 - $f\ x = x$
- Passing x to a linear function g and using the result exactly once in the same fashion.
 - $f\ x = g\ x$

Linear Types

What is **using exactly once**? For a function f with arguments x :

- Returning x unmodified.
 - $f\ x = x$
- Passing x to a linear function g and using the result exactly once in the same fashion.
 - $f\ x = g\ x$
- Pattern-matching on x and using each argument exactly once in the same fashion.
 - $f\ x = \text{case } x \text{ of}$
 $\text{Pair } a\ b \rightarrow a + b$

Linear Types

What is **using exactly once**? For a function f with arguments x :

- Returning x unmodified.
 - $f\ x = x$
- Passing x to a linear function g and using the result exactly once in the same fashion.
 - $f\ x = g\ x$
- Pattern-matching on x and using each argument exactly once in the same fashion.
 - $f\ x = \text{case } x \text{ of}$
 $\text{Pair } a\ b \rightarrow a + b$
- Calling it as a function and using the result exactly once in the same fashion.
 - $f\ x = x + 1$
 $g = \text{let } k = f\ 0 \text{ in } f\ k$

System-level Functional Language (SLFL)

The point of our thesis will be to create a compiler for a SLFL

We want to add several higher level concepts such as:

- Closures
 - Allows lambdas to capture variables from their environment

```
fun :: Int -> (Int -> Int)
fun x = \y -> x + y -- x is captured here
```

- Records
 - Data types with named fields. Pretty simple
- Recursive & Contiguous Data Types
 - Trees, linked lists etc and Vectors/Arrays
- Laziness

How will the language be evaluated?

Objectively evaluating languages is hard, but some things can be done!

- Performance:

Programs will be written in another system-level language (C etc) and SLFL

Results will be compared based on execution time and memory usage

- Binary size:

Small binaries are good for portability

Our thesis will not focus a lot on this, but it is an interesting metric nonetheless

Related Work

- Lilac: a functional programming language based on linear logic [1]
 - Linear type system
 - High-level
 - None or few optimizations
- Linear Haskell: practical linearity in a higher-order polymorphic language [2]
 - Linear type system for Haskell
 - High-level
 - None or few optimizations
- Towards a practical execution model for functional languages with linear types [3]
 - Last year's master's thesis
 - Virtual machine
 - Untyped

Why us?



Why us?

- All FP courses
- All language development courses
- Made a functional programming language for our bachelor thesis
- Good teamwork
- Motivated

Risk assessment and proposed mitigation

- Too complex
- Impossible task

Mitigation

- Work hard!

References

- [1] I. Mackie, “Lilac: A functional programming language based on linear logic,” *Journal of Functional Programming*, vol. 4, no. 4, pp. 395–433, 1994.
- [2] 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.
- [3] F. Nordmark, “Towards a Practical Execution Model for Functional Languages with Linear Types,” 2024.