System-level Functional Programming with Linear Types

Sebastian Selander and Samuel Hammersberg

System level programming today

In todays 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!

But does not functional programming require some sort of garbage collector?

No! We can use linear types!

Yes, we could use linear types as well

Linear Types

Every variable must be used **exactly once**

- Linear arrow: $\multimap$
- Normal arrow: $\longrightarrow$

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

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

- Now `append` can mutate `ys` in a safe manner

Linear Types

For a function `f` with arguments `x` using **exactly once** means:

- Returning x unmodified.
  - `f x = x`

Linear Types

For a function `f` with arguments `x` using **exactly once** means:

- 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

For a function `f` with arguments `x` using **exactly once** means:

- 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 = case x of
      Pair a b -> a + b
    ```

Linear Types

For a function `f` with arguments `x` using **exactly once** means:

- 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 = case x of
          Pair a b -> a + b
    ```
- Calling it as a function and using the result exactly once in the same fashion.

  -
    ```
      f x = x + 1
      g = let k = f 0 in f k
    ```

System-level Functional Language (SLFL)

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

While the language is a system-level language, we want to add several higher level concepts such as:

- Closures
  - Allows lambas 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:
  Simple programs will be written in another system-level language (C etc) and SLFL
  Programs will be compared based on execution time and memory usage
- Binary size:
  A system-level language should ideally produce small binaries 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?

Both of us have a lot of experience when it comes to language development *and* functional programming.

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

Risk assessment and proposed mitigation

- Too complex
- Lack prerequisite knowledge

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.