Compiler optimization in Pen

@raviqqe

Pen's compiler is 4.5 times faster!

- Since the last meetup

Experiment

- Program: `pen compile-prelude` subcommand
- File: `packages/prelude/Map.pen`
    - Runtime module for the built-in map type
    - ~600 lines

	Latency (ms)
Before	358
After	80

What made the compiler faster?

- Code
  - Imperative vs. functional
  - Read vs. write optimized data structures
- Data
  - Minimizing data
  - Sharing data
  - "Compressing" IR

Imperative vs. functional

- Transformation for functional languages often written in a functional way.
  - In papers and books
- It's faster to run imperative algorithms with destructive data structures in Rust.
  - Sometimes, codes get even shorter and more concise.
- Iteration over recursion
  - No tail call elimination in Rust 😭

Examples

- CPS (continuation passing style) transformation
- Type conversion

Read vs. write optimized data structures

- Hash maps are often used to represent variable scopes and their types.
- They are sometimes write heavy.
    - Data is modified more often than it is read.

Examples

- Using lists as maps to track variable types
    - `List (String, Type)`
    - It was faster than using persistent data structures.
        - `list < hash map < persistent hash map`

Minimizing data

- Minimize `enum`s.
  - If only a member is too big, the enum gets also big.
  - There would be too many empty data in a collection of them.
  - Box large members.

Examples

- `Expression` in MIR
- `Instruction`, `Expression`, and `Type` in F--

Sharing data

- Use `&T` if possible.
- Use `Rc` if necessary.
    - Sometimes, cloning is the only option.
    - e.g. borrowing instructions' result names while modifying the instructions themselves
- Even creating `String`s is slow when there are too many of them.

Examples

- Collecting types of local variables
- Collecting free variables of continuations

"Compressing" IR

- It's better to "decompress" IR (intermediate representation) later.
  - Function inlining
  - Reference count operations
    - e.g. clone, drop
- Split those common codes into functions in IR.
- All the later passes get slower by the increased data size.
- LLVM handles the "decompression" anyway.
  - Function inlining
  - CSE

Future work

- Apply those methods everywhere.
- Slow type canonicalization in HIR
- `Rc` all the things (?)
- Function passes
    - Is it better for L1 and L2 caches?