disp Documentation

Yusuke Tsutsumi

Jun 16, 2019

Contents

Disp is a programming language, with the goal of being a practical choice as a single programming languages across an organization.

This goal manifests itself as several sub-goals:

- good performance for many use cases: reduces the need of learning a second, faster language once your primary one hits performance limits.
- upgrading large codebases easily: reducing cost for major refactors, managing code across multiple code repositories.
- maintainability and readability: large organizations will face some level of developer churn, and the person who originally wrote the code may no longer be there.

The specific features of Disp include:

- homoiconic syntax: code is represented using standard data structures
- macro support: function that can do compile-time syntax evaluation
- compile-time execution: functions can execute on runtime, or compile time

> **Warning:** Disp is not ready for use in any production project! Literally anything about the language can change until 1.0

You can see a recent code example here (solving the qualifier problem a for Google Code Jam 2008):

```
let n (int (read-line))
let i 0
while (not (eq i n))
        let s (Int (read-line))
        let possible-engines {}
        let j 0
        while (not (eq j s))
                let e (read-line)
                add possible-engines e true
                let j (+ j 1)
        let q (Int (read-line))
        let j 0
        let switches 0
        let seen-engines {}
        while (not (eq j q))
                let e (read-line)
                add seen-engines e true
                if (eq (count seen-engines) (count possible-engines))
                        let seen-engines {}
                        add seen-engines e true
                        let switches (+ switches 1)
                let j (+ j 1)
        let i (+ i 1)
        print "Case #"
        print i
        print ": "
        println switches
```

To get a better understanding of where Disp shines, it's recommended to read up on the features, and some disp-cases (use cases that inspired disp's design).

CHAPTER 1

Table of Contents

1.1 Installing / Building Disp

Disp is currently only available by building from source. To do so, you will need:

- the Rust programing language tools
- LLVM 7.0 (see your operating system's package manager, there is probably an easier way to build rather than from source.)

Currently the Rust package does an unsavory hack of exposing c abi functions in an executable (so that llvm can read such external functions). In order to accomplish this, additional configuration needs to be set in a .cargo/config file in the $HOME directory, with the following contents:

```
[target.x86_64-unknown-linux-gnu]
rustflags = ["-C", "link-args=-Wl,-export-dynamic"]
```

Once the above is satisfied, you can build Disp by running:

```
cargo build --release
```

And for development:

```
cargo run <file_to_run>
```

1.1.1 Developing Disp

Enabling Debug Features

Sometimes it's helpful to have some more information during execution. That is where debug mode helps:

cargo run –features=debug

Additional output will be printed during disp file compilation, such as:

- grammar parser output
- LLVM IR

1.2 Contributing

If you're interested in joining the development of Disp, fantastic! We are in the process of building a community.

The easiest way to contribute is to add your thoughts to the RFCs: there are a lot of ideas that need further fleshing out, and if you have good evidence or use cases, please leave a comment.

1.2.1 Contributing Code

Contributing code can be done by forking and sending pull requests via Github. If you are looking for something to work on, please see the open issues

Please note that the code is still in a large amount of flux, as implementation details are fleshed out. Any suggestions or improvements are welcome.

1.3 Disp Syntax

Disp's syntax is homoiconic: it is authored with the same syntax that one would use to represent data. The data types that can be represented natively are:

- integers: 0..6+
- strings: ".+"
- lists: [token*]
- expressions: (token*)
- maps: {key: value, key2: value2,+}

1.3.1 Expressions

Expressions are represented in the language as:

```
(function-name arg1 arg2 ...)
```

This will be executed by the compiler, rather than represented as data. Sometimes, it's valuable to defining an expression object without

1.4 Design Discussion

This section is mainly here for brainstorming.

1.4.1 Indentation is important

In clojure, it's very easy to leave additional statements that will be executed on a line above, looking like:

```
(dotimes [i 10] (readline)
  (readline)
)
```

In this case, we will read from stdin twice per loop, as the first statement after the dotimes input is on the same line as the declaration of the dotimes.

It's a lot easier to look at the shape and understand the a block that will be executed is on the next line, and see that block as whole:

```
for i in 10:
  sys.stdin.readline()
  sys.stdin.readline()
```

It's very clear here.

1.4.2 Deep Imports Are Helpful

Rust provides import syntax that allows ones to import specific functions from modules, like:

```
use module::{
  namespace::value,
  namespace2::{value1, value2},
  namespace3
}
```

This allows for:

- imports grouped by root location: helpful to see if an import is already added
- reduces

1.4.3 Use explicit "return" keyword

Rust allows two ways to return a value: either as the last execution in a statement, or with a return statement. The lack of a return statement makes it difficult to search for where termination would occur.

Conversely, it does make it difficult to support single expressions, since that would be easily written as a single expression and it would be clear that it is the return value. Maybe I'm not sure where I stand on this yet.

1.5 Performance

With developer productivity comes a common problem that affects successful software programs: performance. It's often said that performance is often not a primary driver of language choice, and premature optimization is the root of all evil.

However, the optimization process can be significantly more expensive or cheap depending on how the language itself is designed. Examples of language that are difficult to optimize include Python, which require developers to author non-python code to resolve (such as Cython for a Python subset, or C code).

Hard-To-Optimize Language (Python):

- requires coding outside the language to optimize

Easy-To-Optimize Language (Java):

- can get really good performance (1-2x C) before needing to move out of the language
- sticking to the same language enables: * easier developer contribution * simpler build tooling

Compare this to languages which often do not require significant optimization outside the language, such as Java: the VM itself is efficient enough for a wide range of purposes, and thus enables much better developer contribution

1.5.1 Fibers Instead of Threads

System threads work well to distribute CPU-bound workloads, but are expensive when used in a 1-1 ratio with network requests.

Newer programming languages and paradigms have introduced the idea of an eventloop and/or green threading: lightweight threads that are multiplexed on a single thread. These lightweight threads can use operating system level constructs to handle networking, similar to classic threads. The benefits are:

- memory: threads implemented in the application do not require the full stack that system threads too, and thus can be smaller.
-

1.6 Productivity

- writing new code
- understanding legacy code
- resolving bugs with existing code

1.7 Disp Cases

Use Cases, but for Disp! Examples where Disp's properties can be super helpful.

1.7.1 Configuration

At Zillow in 2018, we had a process to pass environment-specific configuration to an application that looked like:

1. grab configuration values for a specific environment
2. render template file with that configuration (typically jinja + yaml)
3. load configuration during application startup

It typically looked something like this:

```
config:
  dns: {{ my_application.dns }}
  port: {{ my_application.port }}
  database:
    connection_string: {{ my_application.db.connection_string }}
```

This works well, but at scale it runs into a couple problems. One is that common configuration ends up being littered across multiple applications, so you end up with boilerplate configuration everywhered.

The other is that people like to get fancy. Jinja as a language supports a broad set of conditional logic, so you can do stuff like completely change the rendered file based on a config value:

```
# fancy set, if / else conditions at the top
{% set svc = my_application %}
config:
  dns: {{ svc.dns }}
  port: {{ svc.port }}
  database:
    connection_string: {{ my_application.db.connection_string }}
  # or a common configuration, such as:
  application_metrics:
    consumer: {{ application_metrics.consumer.host }}
    port: {{ application_metrics.consumer.port }}
    default_namespace: my_application
```

When you have a a lot of boilerplate, at some point a backwards incompatible change comes along, and you need to go change the configuration for everybody. This is a relatively easy task for vanilla yaml: you just need to:

1. iterate through all repos containing this configuration
2. load yaml file
3. rewrite value
4. write yaml file

This loses comments since they aren't part of the data itself, but the application still works as expected.

Unfortunately, you cannot use the Yaml parser to load yaml templatized with Jinja syntax: the brackets {} are parsed by Yaml as an object, as Yaml is a superset of Json. As a result, a simple task like adding a specific value in a specific location in a Yaml hierarchy now becomes an extremely difficult task. In order to do this properly, you would need to:

1. Load the Jinja template into it's data structure representation.
2. Replace variables with placeholders.
3. Hope that the result can be parsed with yaml.
4. If 3 is true, load the values with yaml, and write your value in question.
5. Replace placeholders back with real values.
6. Merge the results with the Jinja data structure you have.

Unfortunately very rarely were all of those aligned, so it becomes effectively impossible to do programmatic updates. Anything that can't be automated at scale means it takes a significant amount of manual effort to accomplish.

How Disp Would Help

The best solution to the above issue is probably decoupling your common components enough that you can avoid this issue (such as having configs live with the library that uses them, and have them source the config directly somehow), but that can often be tricky, and difficult to ensure that no one makes a mistake and causes boilerplate to happen.

The issue with the situation above is that multiple languages are at play. Anytime you mix syntax of two different systems, it is very difficult to write a parser that somehow supports both.

Decomposing the purpose of the two languages above, they are:

1. allow conditional logic and variables (Jinja)

2. structure data so that it can be read by the application (Yaml)

Disp's syntax tree is nothing but lists, maps, and other basic data types: the same ones that would be present in a language like Yaml. When combined with a Disp interpreter, you can execute the code and return the data you're looking for:

```
let val (json (read))
return {
  config: {
    dns: (get (get val "my_application") "dns")
    port: (get (get val "my_application") "dns")
  },
  database: {
    connection_string: (get (get (get val "my_application") "db") "connection_string")
    port: (get (get val "my_application") "dns")
  }
}
```

1.8 Disp's Road to 1.0

Disp is currently highly experimental, and many attributes about it may change. Disp has a specific set of goals it is targetting before reaching 1.0, enumerated here:

- a fully functioning macro system
- a well designed solution for the Array-of-structs and Structs-of-arrays problem
- performance within 1-2x of C for almost all use cases, including high-throughput applications
- a module system
- working solutions to all disp-cases

1.9 Arrays

Unlike c and similar to Rust, arrays are a pairing of two values:

- a pointer to the raw array
- a word that is the explicit length of the array

This convenience was chosen primarily to reduce the amount of work to retrieve the length of said array.

1.10 Working with Strings and Text

As a good primer to working with strings in general, both the Go blog and Rust book have great resources:

- https://blog.golang.org/strings
- https://doc.rust-lang.org/book/ch08-02-strings.html?highlight=string#what-is-a-string

The major takeaway is that common text formats such as UTF-8 are hard to quickly index, unless you scan the whole string beforehand.

To that end, Disp deals with bytes and text in a similar way.

1.10.1 Bytes

Bytes are effectively an array of bytes. They can be indexed quickly O(1). Bytes must be converted to some form of textual representation.

1.10.2 Strings

In the common case for developers, it is not necessary to be particular about the specific string encoding. As such, the "String" type is represented by a common format: UTF-8.

Strings are stored in UTF-8 representation for a couple reasons:

- backwards compatible with ascii, without increasing the size
- can represent all characters in known language character sets

1.10.3 Performance of Indexing of Strings

Due to the variable byte nature of unicode encodings, it's not possible to achieve O(1) indexing of strings without scanning the characters.

Some languages achieve O(1) at the cost of incorrectness in some cases, such as Java (charAt always assumes UTF-16, resulting in incorrect indexes for values that require a larger character set).

As such, it is not possible to achieve fast indexing. As UTF-8 is a superset of ascii, it is possible to optimize the performance of the String if it is identified as being ascii. (at that point it's just finding a byte offset).

In many cases a string being ascii is sufficient, and can result in a significantly higher performance profile O(1) with an offset lookup vs O(n) with a scan:

- developer logging
- API error messages could be reduced to english

All of this implies that the "string" type in disp should be a layer of abstraction that ensures performance in many common cases, although there should still be space for a more custom data type. (potentially allow compile-time reading of literals.

1.10.4 Individual characters in a string are referred to as Runes

It is common nomenclature to call a single byte in an ascii string a "char" or character. As UTF-8 strings can be multiple bytes, we will use the word "Rune" to describe a single UTF-8 character.

1.10.5 No cost to convert from Bytes to a String

It's often value to parse raw bytes into text, as the bytes being read in are guaranteed to be UTF-8.

In that case, it's possible to do a simple type coercion at no runtime cost.

1.11 Why Disp will be a Compiled Language

- enables performance similar to C
- enables use of powerful optimizers such as LLVM.

- enables the standard library to be written in Disp, without incurring performance hits.

1.12 Function Design

For Disp, functions are declared as such:

arguments are represented by dictionaries. The following symbol keys modify the argument behavior:

- :n is the name of the the argument as referred to by the function body and by callers
- :t is the type of the parameter (this is optional. if this does not exist it will be inferred)
- :d is the description of the parameter.

The third argument is the return type.

The fourth argument is the body. In the example above it is the body of the idented block (implicit list)

1.12.1 Keyword Arguments / Default Params Work Wonders for Backwards Compatibility

Backwards-incompatible changes for functions can result in the need for significant refactors and incompatible binary interfaces. A common example of this is Java: as new arguments are added, the interface changes, and requires recompilations and code to be modified, or for the author to support every possible permutation of arguments.

There are two different kinds of interface changes, with common function calls:

Reordering of Arguments

If a positional argument is re-ordered, the argument types change, and as a result every caller must modify their calls.

This can be avoided in languages such as Python, but using the name-value pair on every function call:

```
call_functions(argument_1=foo, argument_2=bar,...)
```

In practice, most library authors can maintain the order of positional arguments.

Adding New Arguments

Oftentimes it is desired to modify function signatures by passing new arguments. This can happen in the case of new flags.

Languages that support optional arguments can add new arguments with ease, as you just provide a default value. By not requiring new arguments to be passed, old callers will work and backwards compatibility is preserved.

1.13 Further Reading and Resources

- Object-Oriented Programming is Bad: https://www.youtube.com/watch?v=QM1iUe6IofM&t=628s
-

1.14 Why Disp is not Object Oriented

Disp does not expose constructs for object oriented programming. Object orientation is itself the combination of multiple separate features of a programming language. Disp will attempt to provide the primitives necessary to enable something similar to object orientation, so one can still benefit from such functionality.

1.14.1 Polymorphism

1.15 Structs

Similar to c or Rust, Disp will support native structs. Rationale:

- enables compact in-memory representations
- compact in-memory representations ensures good cache locality by keeping object sizes low

CHAPTER 2

Features

2.1 Comments in the AST

The AST also includes comments as a first-class type. This is important, as it allows for:

- ast transformations to preserve comments, including comment location
- enables extract of comment information from the AST, such as for building of documentation

2.2 Homoiconic Syntax

A language is considered homoiconic if the language itself can be represented with primitive data structures.

Disp satisfies that, as the language itself can be represented with only the following data structures:

- lists
- maps
- TBD: sets

2.2.1 Easy to Transform

By using simple data structures, it reduces the complexity of transforming the ast in various ways. For example, one can easily add a match condition by reading in a map and adding a key-value branch:

```
// add false to the syntax tree.
macro! add-false-handler [map-foo]
  add map-foo false
```

2.3 Macros

As with other lisps, Disp supports macros. Macros are evaluated compile time and resolve to an ast that is then evaluated by the compiler.

For example, one could create something equivalent to an 'unless' keyword by writing:

```
macro! unless [conditional] body
        ' while (not conditional) body
```

This will expand:

```
unless (eq i x)
        print i
        mut i (+ i x)
```

into:

```
while (not (eq i x))
        print i
        mut i (+ i x)
```

2.3.1 Design Decisions

Why are macro invocations not prefixed with a "!"?

A compiled macro can be argued as a bang expression, since it invokes behavior on compile time (transforming the arguments into a different syntax tree).

There rationale for making it look like a regular function is:

- **macros should be seen as equivalent to builtins, which are not** prefixed with a bang.
- **bang symbols imply that the full statement will be executed on compile** time. In the case of macros, the macro expansion will be invoked, but the body of the code will not.
- this also enables execution of a macro, and it's contents, during compile time. so both of these situations are possible:

```
# runtime evaluation
let x (unless true (print bar))
# compile time evaluation
let y (unless! true (print bar))
```

2.4 Type Inference

Similar to ML and Haskell, Disp supports type checking using type inference, enabling static type validation without the need to specify types signatures in most situations.

For example, you can define a function without types:

```
let p (fn [arg] [(print arg)])
p "foo"
p 10
```

Note that there are two different types passed into "p": a string, and an integer. In this situation, the compiler knows that the print function supports both strings and integers, and will generate separate functions that take their respective types.

On compilation time, disp stores this type of information on all functions, so this relationship is preserved regardless of the functions that are used.

This enables behavior similar to duck-typing, But retaining the ability to type check on compile time. For example, if a function called in the body of a parent function does not support the type in question, the compiler will raise an error:

```
mut add (fn [l r] [(+ l r)])
// this will work
println (add 1 10)
// this will raise a type check error on compilation,
// since the "+" function does not support strings
println (add "foo" 10)
```

CHAPTER 3

Disp-Cases

CHAPTER 4

Indices and tables

- genindex
- modindex
- search