DreamLisp - A Lisp-2 with Modules, Lazy Evaluation and Asynchronous Programming

JASEEM VALIYA VALAPPIL
Department of Mathematics, University of York, UK
(e-mail: jvv502@york.ac.uk*)*

Abstract

DreamLisp is a Lisp dialect with modules, lazy collections and is a Lisp-2 with an unhygienic macro system. DreamLisp interpreter is implemented in Objective-C and as such can run on macOS and iOS. It is tail recursive, uses ARC instead of garbage collection with immutable data structures, has asynchronous communication using notifications taking advantage of the underlying Objective-C runtime and Foundation library. In the experimental version, we can define Objective-C classes and create objects at runtime from the REPL and interop with those from DreamLisp. The source code for the language interpreter is open source and is available at https://github.com/jaseemvv/dreamlisp.

1 Introduction

The ability to program apps for iOS and macOS in a functional language like Lisp would provide an attractive alternative to existing languages like Objective-C and Swift. The advantage of using Lisp is that it provides a REPL where developers can quickly experiment with ideas and the macro system is invaluable. The ability to quickly view the UI changes using the REPL provides a better experience in developing apps. In that respect, we can have an interpreter and a compiler were we can use the interpreter for quick prototyping and the compiler to translate the code to an intermediate language (Objective-C or Swift) and then to the native code using the native compiler for the platform. This paper describes the interpreter.

2 Language

DreamLisp is a Lisp-2 and has semantics closer to Clojure more than Common Lisp, but is a dialect of its own. DreamLisp introduces a module system inspired by Lisp Flavoured Erlang (LFE) and uses MFA (Module, Function, Arity) notation for representing functions. Internally variables, symbols etc. also uses module and arity based notation to keep the implementation consistent. DreamLisp also implements lazy sequences that work with higher-order functions. Function calls use pattern matching based on arity. DreamLisp has a macro system similar to that of Clojure with `gensym` for unique symbol generation as the macros are unhygienic. Macros are compile-time only. The implementation emphasis language usability making errors more human friendly and readable. To have a REPL that does UI updates dynamically, we choose the implementation language to be Objective-C as it is more dynamic and has flexible runtime capabilities which will enable us to create classes and objects at runtime.

3 Data Model

Lisp data structures are objects within Objective-C. For example, a hash-map in DreamLisp is implemented as `DLHashMap` class (referred to as a DL type). These data structures can be manipulated directly from Objective-C which is trivial but manipulating them from DreamLisp is not intended though possible. There are methods to convert between DL and NS types so that we can interop with Objective-C methods seamlessly.

4 Module System

We can define a module from the REPL as well as load from a file. All functions that come after the module definition belongs to that module. Functions from another module can be invoked either by using the fully qualified form which is of the form `module:function-name/arity` or by importing it into the module using the import special form.

All functions defined in a module is by default private to that module. For other modules to use a function from a module, those functions must be exported. We can only invoke exported functions of a module and there are no ways to get around it. Functions with the same name and different arity are considered different.

Below is an example that shows a module called utils in a file `utils.dlisp`, that defines two functions `collatz/1` and `collatz/2`. We export only the `collatz/1` function.

```
; utils.dlisp
(defmodule utils (export (collatz 1)))

(defun collatz (n)
  (collatz n []))

(defun collatz (n acc)
  (if (not= n 1)
    (if (even? n)
      (collatz (/ n 2) (conj acc n))
      (collatz (+ (* 3 n) 1) (conj acc n)))
    (conj acc n)))
```

Say we load this file into the REPL, we can invoke the `collatz/1` function in a fully qualified form as:

```
λ user> (utils:collatz 21)
[21 64 32 16 8 4 2 1]
```

We can import the function from another module. If the local module has a function with a similar name, it gets replaced.

```
; greet.dlisp
(defmodule greet
  (export all))

(defun hello () "hello")

; mtest.dlisp
(defmodule mtest
  (export all)
  (import (from utils (collatz 1))
          (from greet (hello 0))))

(defun say-hello () (hello))

(defun main () (collatz 21))
```

Now we load all the modules and invoke the functions. DreamLisp does not load any dependent modules on its own. We need to load them into the env before executing the function.

```
λ user> (load-file "utils.dlisp")
[:ok "utils.dlisp"]
λ user> (load-file "greet.dlisp")
[:ok "greet.dlisp"]
λ user> (load-file "mtest.dlisp")
[:ok "mtest.dlisp"]
λ user> (mtest:main)
[21 64 32 16 8 4 2 1]
λ user> (mtest:say-hello)
"hello"
```

Here we load modules in the reverse order of the dependency. Alternately, we can write a function to do this. The `load-file` function loads the file into the current environment and evaluates it. Here the module `greet` exports all functions. The module `mtest` imports `hello` from `greet` and references `collatz` from the module `utils` using its fully qualified form.

Internally, we make use of a fault object when parsing the `defmodule` expression where we defer the association of the import symbols until the function is invoked. This helps us to load the module without causing a "symbol not found" error. The concept of fault is similar to that of Core Data faults. It is represented using `DLFault` and when accessing the import symbol, if it resolves to a fault, the fault is tried to be resolved by replacing it with the actual symbol referenced if found or throws an error.

DreamLisp uses MFA notation for functions. For example, say we have a module `utils` and a function `collatz` with no arguments, then it is represented as `utils:collatz/0` and say we have one argument, it is represented as `utils:collatz/1` and to represent variadic functions, we use `n` which gives `utils:collatz/n`. The `n` arity is internally represented using `-1`. Symbols and variables use the arity `-2` though this notation is not exposed to the user.

The module related functions are described below. The `defmodule/3`, `in-module/1` and `remove-module/1` are implemented as special forms.

`defmodule/3`
This is used to define a module, which can have `export` and `import` expressions with the function name, arity tuples. Using `(export all)` exports all functions in the current module.

```
λ user> (defmodule tree (export (create-tree 0) (right-node 1) (left-node 1)))
tree
λ user> (defmodule stage (export (greet 0)) (export all) (import (from player (sum 1))))
stage
```

`in-module/1`
Changes the current module to the given module.

```
λ user> (in-module "core")
"core"
λ core>
```

`remove-module/1`
Removes a loaded module from the module table.

```
λ user> (defmodule mtest (export all))
```

```
mtest
λ mtest> (remove-module "mtest")
nil
λ user>
```

current-module-name/0
Returns the current module name.

```
λ user> (current-module-name)
"user"
```

module-info/1
Returns the module information which contains the imports, exports, and other metadata.

```
λ user> (module-info "test")
{:description "A module to work with unit tests." :imports [] :internal
[] :name "test" :exports [test:get-test-info/0 test:dec/1 test:inc/1
test:inc-test-count/0 test:is/3 test:update-test-info/1 test:deftest/n
test:testing/n test:is/2 test:run/1]}
```

module-exist?/1
Checks if the given module is loaded.

```
λ user> (module-exist? "test")
true
λ user> (module-exist? "spaceship")
false
```

all-modules/0
Returns all the loaded modules.

```
λ user> (all-modules)
["core" "test" "user" "network"]
```

DreamLisp comes with built-in module `core`, `network` and `test`. The core module contains all functions that work with basic data structures like list, hash map, string and so on. The network module has functions to work with HTTP endpoints. The test module contains functions for writing unit tests and a test runner.

5 Execution Model

There are a global module `user` and custom modules defined by the language itself like `core`, `test` and the user specified modules. Each module has its environment which is represented using `DLEnv`. This class consists of an export table, an import table, an internal table and a symbol table. As the name suggests, the export table contains symbols that are exported by the module, the import table consists of symbols imported by the module, the internal table consists of symbols which are private to the module and the symbol table is used to keep track of objects that uses the same binding name. Depending on the expression and the argument position we pick the right symbol. For example, a function which takes a function as its argument needs to be fully qualified.

```
λ user> (apply core:take/2 3 [3 1 4 1 5])
[3 1 4]
```

Here `apply` takes a function, a variable number of arguments and applies the function to the arguments.

```
λ user> (def fname "ol")
"ol"
λ user> (def lname "ive")
"ive"
λ user> (defun fname () "first-name")
user:fname/0
λ user> (str fname lname)
"olive"
```

Here even though we have symbols with the same name `fname` defined twice, they are different and since `fname` appears in argument position without the MFA format, we pick the variable instead of the function represented by that same name.

The module tables and symbol table is implemented using `NSMapTable` which is more flexible in working with custom objects as key in the table.

6 Lazy Sequence and Evaluation

DreamLisp supports lazy sequence and higher-order functions that work with them. We can also create lazy sequences from normal sequences using the built-in function `lazy-seq`. Collections returned by higher-order functions are lazy by default. The operations are deferred until we force evaluate either by applying other list functions over the lazy sequence like `first/1`, `take/2` or by using `doall/1`. The `doall/1` applies the function associated with the lazy sequence if present to the elements as required to realise the resultant elements in the sequence. It is implemented using the `DLLazySequence` class. The sequence and the function to be applied to the sequence is held by this object. Lazy sequences implement `has-next?/1` and `next/1` functions and execute the function over the sequence until the function condition is satisfied or the entire sequence is processed.

7 Asynchronous Programming

DreamLisp uses notifications for passing messages. It internally uses `NSNotification` to post and observe for notifications. This technique is also used as a callback mechanism for places where Objective-C expects a delegate. This is used in the network-related APIs.

```
λ user> (defun pong (x) (println "pong" x))
user:pong/1
λ user> (add-notification :pong pong/1)
true
```

The `add-notification/2` function is used to listen for a notification. The key is specified as a keyword with the handler function specified in MFA format.

```
λ user> (post-notification :pong "psi")
pong psi
true
```

A message can be send using `post-notification/2` with the notification name as the keyword followed by any data. All handlers registered to listen for a particular notification will be invoked when the notification is received.

```
λ user> (remove-notification :pong)
true
```

The `remove-notification/1` removes all handlers for the given notification.

8 ARC (Automatic Reference Counting)

DreamLisp uses Automatic Reference Counting (ARC) instead of garbage collection. It takes advantage of the host language features where Objective-C itself uses ARC. The implementation does resort to manual reference counting for certain classes like `DLObjc.m`, `DLFileOps.m` to improve the interpreter performance by having more control over the memory release cycles, while other classes have ARC enabled by default.

9 Objective-C Runtime Interop

This is an experimental feature where we can define classes and create objects at runtime from the REPL. The DL data types and NS data types are separate and do not interop directly. Methods that converts to and from DL types to NS types are implemented. We need to deal with this only when we are directly working with Objective-C objects in the REPL. To work with object-oriented programming the language exposes constructs similar to CLOS (Common Lisp Object System).

```
λ user> (defclass Person (NSObject) ((name :initarg :with-name)))
#<class Person>
```

This defines a class `Person` in Objective-C which inherits from `NSObject` and has a property called `name` as a slot with the init method as `- (instancetype)initWithName`.

```
λ user> (def olive (make-instance 'Person :init-with-name "Olive"))
#<object <Person: 0x60200001ded0>>
```

This will create an instance of the `Person` class with the name set to Olive and assigns it to the `olive` variable. The property is represented by `DLSlot`. The class is represented by `DLClass` and the instance by `DLObject`. We can add a method to a class using `defmethod` form. The method definition is represented as `DLMethod`.

```
λ user> (defmethod person-id 'Person (n) (* 3.141 n))
#<method person-id:>

λ user> (person-id olive 21)
65.961
```

Currently accessing the slot variable of an object from a method is not supported. It is left to future implementation.

The Foundation classes generally use the naming convention with an NS prefix. Working with Lisp, we use lisp case. We try to automatically convert from these types. This is implemented using a trie data structure. For conversion between pascal case, camel case and lisp case a trie data structure with prefixes is maintained. We search the trie when a class or method name is encountered. For example, `ns-url-session` is converted to `NSURLSession`. The prefix tree has `NS` and `URL` as separate nodes and the next word's first character is converted into uppercase and concatenated. We can alternately resort to pascal or camel case in which case the automatic case conversion does not take place. Data types are automatically converted from DL type to NS type and vice-versa when working with CLOS based methods.

10 Error Handling and Exceptions

DreamLisp has error handling with `try-catch-throw` construct. An expression can be wrapped in `try` and `catch`. If an exception occurs within the `try` expression, the `catch` block is invoked with the exception. We can raise a custom exception using the `throw` function.

```
λ user> (try (throw {:err "oops"}) (catch e e))
{:err "oops"}
```

11 Discussion

The interpreter shapes the language constructs, implements the specification. We can then write a compiler that converts the lisp code to Objective-C or Swift and then use the native compiler to generate the native code. This way we can ship apps that run on both iOS and macOS that runs natively. The compiler can be written in DreamLisp itself or in any other language.

Further work can be done related to accessing a slot variable of an object. Also, loading dependent module for a loaded module by walking through the current path would be valuable. The Objective-C Interop is still in an experimental phase and can be further enhanced to add more features and to make it more stable.

Bibliography

1. "LFE (Lisp Flavoured Erlang)". Robert Virding. Accessed 12 July 2020. https://github.com/rvirding/lfe
2. "MAL (Make a Lisp)". Joel Martin. Accessed 12 July 2020. https://github.com/kanaka/mal
3. "Clojure". Rich Hickey. Accessed 12 July 2020. https://clojure.org/

Conflict of Interest

None.

References

Bawden, A. (1999). Quasiquotation in Lisp. Partial Evaluation and Semantic-Based Program Manipulation (p./pp. 4-12).

Queinnec, C. (1996). *Lisp in Small Pieces* (K. Callaway, Trans.). Cambridge: Cambridge University Press. doi:10.1017/CBO9781139172974

S. E. Keene. 1988. A programmer's guide to object-oriented programming in Common LISP. Addison-Wesley Longman Publishing Co., Inc., USA.