Kernel#

source code walk-through

Introduction

Hi guys, Kernel# is the name of my latest project to implement a functional programming language. The aim was to demonstrate that such a big undertaking can be done in a clean and modular approach. This document should describe the architecture of this project and can act as an example for general implementation of interpreters.

What is Kernel?

Kernel is based on a doctor thesis by John Shutt (http://web.cs.wpi.edu/~jshutt/kernel.html). He is the inventor of a so called “vau calculus”, which unifies ordinary functions with macros. This idea reduces the amount of primitives for a language to a very small core. The simplicity of Kernel allows for a straightforward implementation. The actual specification of the programming language is called Kernel, which is influenced heavily by Scheme. The language has a more theoretical usage because vau-calculus forfeits many abilities to optimize code. But this is a fair trade-off for its clean design.

Everything is an object

The best entry point for an implementation is the object system, because this part of the interpreter has no dependencies. We can create it “from empty space”.

```
public abstract class KObject
{
    public abstract string Print();
    public abstract bool CompareTo(KObject other);
}
```

This is the summary of the base class for every kernel object. Because Kernel is a dynamic typed language, most of our functions will use KObject as type. We specify two behaviors every type of object need two implement: Printing and Comparison.

There are, all in all, nine different types of objects, which are built-in into the interpreter. More types will be added by modules, but there are unimportant now. All following types can be found under “src/core/”

nil

This type is very simple. It has no state and prints to the string “()”:

```
public class KNil : KObject
{
```

```
    public override string Print(){
        return "()";
    }

    public override bool CompareTo(KObject other){
        return other is KNil;
    }
}
```

inert/ignore

These two types are as simple as nil. The have no state and prints to the strings “#inert” and “#ignore”:

```
public class KInert : KObject
{
    public override string Print()
    {
        return "#inert";
    }
    ...
}

public class KIgnore : KObject
{
    public override string Print()
    {
        return "#ignore";
    }
    ...
}
```

boolean

The next type contains only two distinct values: true and false. The are printing to “#t” and “#f”. The value is stored internally within a bool-property.

```
public class KBoolean : KObject
{
    public bool Value {
        get;
        private set;
    }

    public KBoolean(bool value)
    {
        Value = value;
```

```
    }

    public override string Print(bool quoteStrings)
    {
        return Value ? "#t" : "#f";
    }

    public override bool CompareTo(KObject other)
    {
        return other is KBoolean && (other as KBoolean).Value ==
Value;
    }
}
```

symbol

Symbols are the fundamental mechanism for abstraction, as they enable us to link to other parts of the code. The implementation of symbols is very similar to booleans, but symbols itself are represented as a string. The value of the string is the name of the symbol. the symbol prints itself to this string.

pair

Pairs compose two elements and allow us to build arbitrary data-structures, because pairs can be nested recursively. The implementation contains two fields with the type KObject, with are called Car and Cdr for historical reasons. The printing of a pair may be interesting:

```
public override string Print(bool quoteStrings)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("(");
    KPair cur = this;
    while (true) {
        sb.Append(cur.Car.Print(quoteStrings));
        if (cur.Cdr is KNil) {
            sb.Append(")");
            break;
        } else {
            if (cur.Cdr is KPair) {
                cur = (KPair)cur.Cdr;
                sb.Append(" ");
            } else {
                sb.Append(" . ");
                sb.Append(cur.Cdr.Print(quoteStrings));
                sb.Append(")");
                break;
```

```
            }
        }
    }
    return sb.ToString();
}
```

A *list* is a chain of pairs where the Cdr points to the next pair and where the last Cdr is nil. Such a list will be printing within round parentheses. If the last Cdr is not nil, then it will separated by a dot.

Pairs are the building blocks of Kernel. To find out more about pairs and there usage you can consult any scheme-tutorial.

Important note: Pairs in Kernel# are immutable. After construction the references of Car and Cdr can not be changed. This will not limit the expressiveness of the language, but allows simpler reasoning.

environment

An environment is used to store *bindings*. Every binding consist of a pair of key and value. This is the place where all data is stored. Kernel makes environments first-class and accessible. Furthermore environments build an hierarchy. Every environment has a parent environment. This is important for lookups because if a binding can't be found in an environment, then the parents environment will be searched through. The top level environment has no parent. Environments can't be printed.

applicative/operative

These types are the representation of a first-class functions and macros. They both derive from KCombiner. An applicative is basically a wrapper around an operative. The difference arise only in the evaluator. The heavy part is done in the operative. They contain a reference to three KObjects and to an environment, in with the operative will be executed (lexical scoping).

We will not dive deeper into the principles of Kernel. Instead we will continue with other next important part of the language:

Parser: from text to objects

The parser reads text input transform it to objects we defined earlier. Therefore the parse only depends on the object system. The parser can be found in “src/parser/”.

TokenDefinition

We begin with defining the tokens. Every token will be recognized by a regular expression. The definition with the highest precedence will be chosen. Some values, e.g. whitespace, can be ignored.

Token / TokenStream

In the first step the parser splits the text into tokens. These tokens are stored in a TokenStream.

TextHandler

Most elements are text elements. The are processed by a TextHandler, which contains a callback to a read function. TextHandlers are selected by precedence and they take care booleans, ignore, inert, boolean, empty environments and symbols.

The chars '(', ')' and '.' are reserved for lists, whitespace and comments will be left out.

CPS crazyness

This part is very complex, but is necessary to support tail recursion. They can be found in the folder “src/CPS/”. Theses classes are independent and generic.

Continuation

This object stores a call to the next function on the call stack. It is similar to a return.

RecursionResult

Instead of returning values, an object is returned which contains a continuation object. This will be call on the next run.

CPS

This is the underlaying trampoline for all function calls. With this technique, we can build infinite chains of function calls without exhausting the stack.

Evaluator

Here the grammar of the language kicks in.

Modularization

Two classes will help us making the system modular.

POperative

This is the outward interface to implement custom primitives.

Module

Manager for library code and dependencies.

Interpreter

The interpreter combines the parser with the evaluator and the module system. It manages loading of modules, parsing of text and then there evaluation. Furthermore it provides a repl. The entry point of the program looks like this:

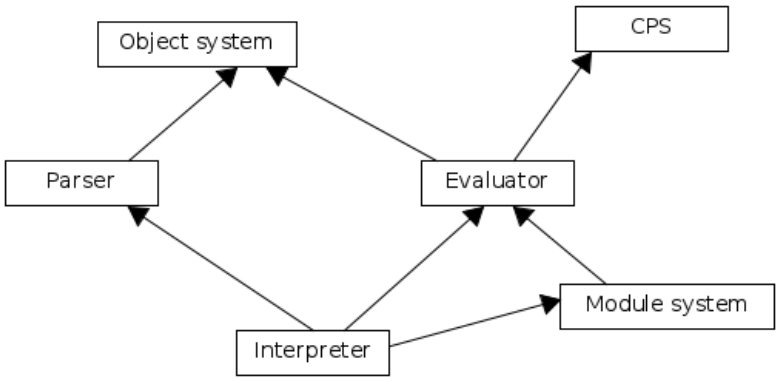
```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to Kernel#harp! Call (exit) to
quit.\n");

    Interpreter.LoadModule(new FastWrapModule());
    Interpreter.LoadModule(new CoreModule());
```

```
    Interpreter.LoadModule(new StringModule());
    Interpreter.LoadModule(new ConsoleModule());
    Interpreter.LoadModule(new ExceptionModule());
    Interpreter.LoadModule(new LoadModule());
    Interpreter.LoadModule(new TypesModule());
    Interpreter.LoadModule(new OperatorsModule());
    Interpreter.LoadModule(new EnvironmentModule());
    Interpreter.LoadModule(new ContinuationModule());
    Interpreter.LoadModule(new EncapsulationModule());
    Interpreter.LoadModule(new NumbersModule());

    Interpreter.REPL();
}
```

The interpreter loads all necessary modules and then runs the REPL. This is the structure of our interpreter so far:



This is the dependency structure of the core library. With the FastWrapModule we can use a primitive wrap which is significantly more efficient: