1 Introduction

This is a guide on how to use the Krivine Abstract Machine, which is built using the Haskell language.

1.1 How to run

As this is implemented in Haskell, the language will have to be installed using Chocolatey. Of course, it would be nice to have an easier way to install Haskell but the instructions are clear and concise, and can be found at this link: `https://www.haskell.org/platform/`.

Along with the required packages for the language to run, you will also receive GHCi, which is the compiler. To run a Haskell file, navigate to the directory that the file is in using command prompt. When you arrive at this directory, simply type the following command: `ghci filename.hs`. The functions that are available to use will be described in later sections.

There is also an option of running the code on WinGHCi, a GUI that makes it easier to do so than on the command line. The link for instructions can be found here: `https://ryanjohnson.website/install-guide-for-winghci-2522/`. Note that WinGHCi is only compatible with Windows and may require Visual Studio to be installed.

1.2 Prerequisites

The prerequisites for this project are:

1. Lambda-terms consist of three constructors: variable, lambda and apply. These match the three cases of the definition of a lambda-term: M ::= x | λ.x | M M
2. A function, known as "pretty", that converts terms to lambda-terms by representing λ as a \
3. An example lambda term, λa.λx.(λy.a) x b, is provided for testing. Bear in mind that on the interpreter, it will show as: `\a. \x. (\y. a) x b`, as touched on in prerequisite
4. Some functions that implement substitution, renaming and fresh variable generation

2 What is the Krivine Abstract Machine?

The Krivine machine is a means of performing λ-calculus operations using call-by-name, as opposed to the call-by-value used in a traditional λ-calculus reduction. The machine uses "weak head reduction" to execute λ-calculus. This

means that there must be a feasible redex to reduce at the start of a λ-term. If that is not the case, then no computation occurs.

2.1 Is it better than direct beta-reduction?

For context, it is best to check out the Lambda Calculus Interpreter here: `https://github.com/manvirdx/LCI` before using the Krivine Abstract Machine. Some of the functions present in this project have been taken from the interpreter and are pivotal in the correct functioning of this machine.

It differs from the implementation of beta-reduction in this interpreter in one major way; it is more efficient. This is because the direct implementation of beta-reduction repeatedly traverses a term to look for redexes or variables to rename. Of course, this is unsurprising given the various reduction strategies available. On the contrary, the Krivine Abstract Machine checks only the head of the term and is therefore less computationally intensive.

This improvement is helped by two restrictions. Firstly, the machine assumes that terms are closed (i.e. there are no free variables. Secondly, it uses a reduction strategy known as weak head reduction. The benefits of such a strategy is that it can't produce a reduct where variable capture can happen, resulting in less renaming. Funnily enough, weak head reduction is a restriction in normal order reduction, as the latter doesn't perform reductions in an abstraction, or in the argument N of an application MN.

Indeed, these redexes do need organisation. Therefore, they are stored in a stack to allow direct access to the head redex without the need of traversing the term at all.

3 The Partial Abstract Machine

To show the concept of streamlining redex searching, a partial abstract machine is first developed. This starts by defining a state, which is a pair of a term and a stack of terms represented by this: $s ::= \star \mid N \cdot s$. Here, N represents the term, s represents the stack and $\star$ represents some empty stack.

```
*ghci> state1 :: PState
(\x. \y. x[Yes,No])
```

Now that the concept of a partial abstract machine and a state type have been defined, it is time to discuss the transition steps that make this. Two terms, `term1` and `term2`, have been provided to demonstrate the effectiveness of these functions.

3.1 Start state

A start state is a pair $(N, \star)$, derived by the transformation of some term N. As in the definition of the state type, $\star$ represents an empty stack.

```
*ghci> p_start term1
((\x. \y. x) Yes No, [])
```

3.2 Transitions

The transitions are simply the intermediates between the start and final state of some term N. For transitions to happen, the states must be of one of the following two formats to produce feasible mappings.

$$(\lambda x.N, M \cdot S) \mapsto (N[M/x], s)$$
$$(NM, s) \mapsto (N, M \cdot S)$$

For the first format, substitution is used to eliminate the variable name x. For the second, M is added to the stack. The function `p_step` checks these two formats.

```
*ghci> p_start term1
((\x. \y. x) Yes No, [])
*ghci> p_step it
((\x. \y. x) Yes,[No])
*ghci> p_step it
(\x. \y. x, [Yes,No])
*ghci> p_step it
(\x. \y. x, [Yes,No])
*ghci> p_step it
(Yes, [])
```

3.3 Final state

Of course, reduction needs to end eventually. Hence, a term will need to reach a final state to do so. Such a term needs to be in one of two forms to count as a final state: $(\lambda x.N, \star)$ or (x, s). This is done with the function `p_final`. When checking that a state is final, a boolean is returned. If it returns false, then it will need at least one more step.

```
*ghci> p_start term1
((\x. \y. x) Yes No,[])
*ghci> p_step it
((\x. \y. x) Yes,[No])
*ghci> p_step it
(\x. \y. x,[Yes,No])
*ghci> p_step it
```

```
(\x. \y. x,[Yes,No])
*ghci> p_step it
(Yes,[])
*ghci> p_final it
True
*ghci> p_final (p_start term1)
False
```

3.4 Running the machine

There are two major iterations in this. Firstly, to speed up the process of reducing a term, the function `p_run` is provided to perform all steps in one call (similar to the `normalize` function in the lambda calculus interpreter). The function performs the following steps:

1. Initialise a term N as a start state
2. Output the current state
3. Check if it's final
    - If it's final, then end
    - If it's not final, then apply a transition step and repeat steps 2 and 3

```
*ghci> p_run term1
((\x. \y. x) Yes No,[])
((\x. \y. x) Yes,[No])
(\x. \y. x,[Yes,No])
(\x. \y. x,[Yes,No])
(Yes,[])
```

However, there is one more step required; the stack needs to be converted back into an application sequence. This is done by expanding `p_run` with a subsidiary function, `p_readback`, that is called when the term is at a final state. To show the term derived by `p_readback`, `p_run` prints it out when a final state is reached. Note that `p_readback` is unusable on its own.

```
*ghci> p_run term2
((\b. (\a. \x. (\y. a) x b) Yes) (\z. z) No,[])
((\b. (\a. \x. (\y. a) x b) Yes) (\z. z) [No])
(\b. (\a. \x. (\y. a) x b) Yes,[\z. z,No])
((\c. \a. (\a. c) a (\z. z)) Yes,[No])
(\c. \a. (\a. c) a (\z. z) [Yes,No])
(\b. (\a. Yes) b (\a. a),[No])
((\a. Yes) No (\c. c),[])
((\a. Yes) No [\c. c])
(\a. Yes,[No,\c. c])
(Yes,[\c. c])
Yes (\c. c)
```

4 The Full Machine

The Partial Abstract Machine is able to demonstrate a streamlined way of finding a redex. Now, the machine can be expanded to streamline substitution in a similar way. Rather than performing substitutions in term N immediately, the term and replacement (i.e, what is to be substituted in) are kept separate. This is done by defining a pair (N, E), also known as a closure, where N is a term and E is a set of substitutions, also known as an environment and initialised as a stack of triples.

$$E ::= \star \,|\, (x, N, E) \cdot E$$

The triple is formatted such that it represents the substitution of a variable x by a closure (since the pair (N, E) is a closure). Substitutions are carried out in ordered to retrieve a term from the closure.

There are four possible substitutions of the form $[\![N]\!]_E$:

$$[\![x]\!]_* = x$$
$$[\![x]\!]_{(y,N,E)\cdot F} = \begin{cases} [\![N]\!]_E & \text{if } x = y \\ [\![x]\!]_F & \text{otherwise} \end{cases}$$
$$[\![\lambda x.N]\!]_E = \lambda x.[\![N]\!]_{(x,x,\star)\cdot E}$$
$$[\![NM]\!]_E = [\![N]\!]_E [\![M]\!]_E$$

Now the Krivine Machine can be defined in the following functions.

4.1 Defining data types

To start off, two data types have been initialised. The first, `Env`, defines an environment. The second, `State`, defines a state. A state is represented as a triple (N, E, S), where N is a term, E is an environment (such that (N, E) makes a closure) and S is a stack of closures, such that:

$$S ::= (N, E) \cdot S$$

Furthermore, a display function (which is not usable) is added to display `Env` as a list of triples and `State` as a triple. Three example terms have also been provided to show how states are displayed.

```
*ghci> state2
((\x. x) y,[("y",\z. z,[])],[])
*ghci> state3
(x x,[("x",\x. x x,[])],[])
*ghci> state4
(\y. x,[],[(z,[("z",\a. b,[("b",c,[])])])])
```

4.2 Start state

The `start` function sets up a term N and turns it into a start state, which is a state of the form $(N, \star, \star)$

```
*ghci> start term1
((\x. \y. x) Yes No,[],[])
```

4.3 Transitions

Like in the Partial Abstract Machine, the transitions are simply the intermediates between the start and final state of some term. For transitions to happen, the states must be of one of the following four formats to produce feasible mappings. Note that in the second format, $x \neq y$.

$$(x, (x, N, F) \cdot E, S) \mapsto (N, F, S)$$
$$(x, (y, N, F) \cdot E, S) \mapsto (x, E, S)$$
$$(\lambda x.N, E, (M, F) \cdot S) \mapsto (N, (x, M, F) \cdot E, S)$$
$$(NM, E, S) \mapsto (N, E, (M, E) \cdot S)$$

The function `step` carries out a single transition step at a time.

```
*ghci> step state2
(\x. x,[("y",\z. z,[])],[(y,[("y",\z. z,[])])])
*ghci> step it
(x,[("x",y,[("y",\z. z,[])]),("y",\z. z,[])],[])
*ghci> step it
(y,[("y",\z. z,[])],[])
*ghci> step it
(\z. z,[],[])
```

4.4 Final state

A term will need to reach a final state for reduction end. Such a term will need to be in one of the two following formats: $(\lambda x.N, E, \star)$ and $(x, \star, S)$. This is done with the function `final`. As in the partial machine, a boolean is returned when checking for a final state.

```
*ghci> step state2
(\x. x,[("y",\z. z,[])],[(y,[("y",\z. z,[])])])
*ghci> step it
(x,[("x",y,[("y",\z. z,[])]),("y",\z. z,[])],[])
*ghci> step it
(y,[("y",\z. z,[])],[])
*ghci> final it
False
*ghci> step it
```

```
(\z. z,[],[])
*ghci> final it
True
```

4.5 Running the machine

To speed up the process a reducing a term, the function `run` is provided to perform all reduction steps in one function call (similar to `p_run` in the partial machine). The `run` function performs the following steps:

1. Initialise a term N as a start state
2. Output the current state
3. Check if it's final
    - If it's final, then end
    - If it's not final, then apply a transition step and repeat steps 2 and 3

```
*ghci> run term1
((\x. \y. x) Yes No,[],[])
((\x. \y. x) Yes [],[(No,[])])
(\x. \y. x,[],[(Yes,[]),(No,[])])
(\y. x,[("x",Yes,[])],[(No,[])])
(x,[("y",No,[]),("x",Yes,[])],[])
(x,[("x",Yes,[])],[])
(Yes,[],[])
```

4.6 Reading back

The final step in the way that the Krivine Abstract Machine performs reductions is converting the stack back into a list of applications. This is done while interpreting each closure as a term $[\![N]\!]_E$. An easier way of putting this is converting a state back to a term. The `readback` function implements this.

```
*ghci> readback state2
(\x. x) (\z. z)
*ghci> readback state3
(\x. x x) (\x. x x)
*ghci> readback state4
(\y. x) (\a. c)
```

An interesting feature of `readback` is that it can be called inside `run`. To accommodate this further, `run` is changed to print out the corresponding term when a final state is reached. It's probably a good time to discuss infinite reduction loops. If that occurs, simply press Ctrl+C to kill the function call.

```
*ghci> run (readback state2)
((\y. x) (\a. c),[],[])
(\y. x,[],[(\a. c,[])])
(x,[("y",\a. c,[])],[])
(x,[],[])
x
```

5 Outro

This document has provided the instructions to the Lambda Calculus interpreter, implemented in Haskell.

5.1 Alternative types

When defining data types at the start of the machine, there is a way of doing this through the use of a supplementary data type, `Closure`. This would, of course, represent the pair (N, E). While such an addition would make implementation look cleaner, the outputs would be slightly different (most outputs will have an extra empty [] at the end). However, the number of steps will remain the same.

```
*ghci> term1
(((\x. \y. x) Yes No,[]),[])
(((\x. \y. x) Yes,[]),[(No,[])])
((\x. \y. x,[]),[(Yes,[]),(No,[])])
((\y. x,[("x",(Yes,[]))]),[(No,[])])
((\x,[("y",(No,[])),("x",(Yes,[]))]),[])
((x,[("x",(Yes,[]))]),[])
((Yes,[]),[])
```

5.2 Final words

If there are any enquiries, then please highlight them on Github. Projects implemented that are lambda-calculus based, may continue to be uploaded in future.