Assignment 4

Advance Programming

Name: Kazi Sohrab Uddin Titu, Group Number: 58

1 Introduction:

This assignment consists of some features like to handle both pure and IO-based evaluations, including key-value store operations and transaction effect. I estimate the quality of my solution to be quite satisfactory, as the core functionalities work as expected; however, some edge cases and error handling scenarios may not have been fully addressed. I have tried to ensure coverage of different cases. Though there are some edge cases that could be refined further but I tried to cover them up. To run test cases, run cabal test in the terminal.

2 Task: The TryCatchOp Effect

Solution Functionality:
The Try-Catch mechanism has been implemented, enabling the interpreter to handle errors during the evaluation of expressions. When an error occurs in the first computation (m1), control is transferred to the alternative computation (m2), allowing for graceful error recovery. The TryCatchOp has been implemented to facilitate error handling in the interpreter. This mechanism allows for recovery from failures without crashing the interpreter. However, the implementation currently lacks isolation for key-value store effects made during m1, which may lead to unexpected behavior in subsequent evaluations.

Failures or Limitations:
The current implementation does not correctly isolate key-value store effects made during the execution of m1. If m1 performs operations that affect the key-value store before failing, those changes remain in effect when evaluating m2, potentially leading to unexpected results.

Potential Issues:
A significant concern is ensuring that the key-value store state reflects only the successful computations. This may necessitate the addition of a rollback mechanism to restore the key-value

store to its previous state if m1 fails. Addressing this issue is crucial for maintaining the integrity of the interpreter's state and ensuring consistent behavior across evaluations.

3 Task: Key-value Store Effects

Solution Functionality:
My implementation of key-value store operations is functional. The operations KvPut and KvGet work as expected, allowing values to be stored and retrieved by their associated keys. The task was solved by extending EvalOp with KvGetOp and KvPutOp and updating the Functor instance to handle these new operations. I have written a replaceState function which is designed to update or insert a key-value pair in a state representation, which is typically a list of pairs. The TryCatchOp effect is evaluated, where if the first operation (m1) succeeds, its effects (like updating the store) persist. However, if m1 fails, the changes made to the key-value store are either discarded in a pure interpreter (no visibility of changes in subsequent operations) or can remain visible in an IO-based interpreter, depending on its transaction handling. This behavior highlights the difference in state management between pure and IO-based interpretations, particularly how failures impact the visibility of changes in the key-value store.

Failures or Limitations:
However, there are some failure cases to note:

- **Invalid Key Retrieval**: The implementation fails when attempting to retrieve a value using a non-existent key, returning an appropriate error message.
- **Overwriting Values**: While the system can overwrite existing values, ensuring that the previous value can be restored upon transaction rollback has proven to be a challenge.

Potential Issues:
The primary issue lies in how state management is handled during transactions. Specifically, it appears that the rollback mechanism does not always restore the previous state correctly, leading to inconsistencies in the data.

4 Task: TransactionOp effect

Solution Functionality:
The transaction feature allows for the execution of multiple key-value operations in a single atomic operation. If any operation within the transaction fails, the entire transaction rolls back,

preserving the state of the database. The copyDB function is used in the transaction task to create a temporary copy of the key-value store before executing a transaction. This allows the interpreter to test the effects of the transaction in isolation. If the transaction succeeds, the changes are applied to the main database. If it fails, the temporary copy is discarded, ensuring that the original database remains unchanged. This approach provides a safe way to handle operations that may alter the database state, supporting rollback functionality.

Failures or Limitations:
I have tried to check edge cases also and found that there are some failures limitations like:

- **Error Propagation**: Currently, the error messages returned by failed transactions do not provide enough context for debugging.
- **Transaction Rollback Failure**: We encountered challenges in implementing rollback functionality for failed transactions in the IO-based interpreter. The lack of a working rollback mechanism meant that any failed transaction could leave the database in an unexpected state, hindering reliable error handling.

Potential Issues:
The issue may stem from the handling of the temporary database. Ensuring that all operations are correctly isolated, and that the original database state is restored requires further refinement.

5 Questions:

1. Consider interpreting a TryCatchOp m1 m2 effect where m1 fails after performing some key-value store effects.

(a) Is there a difference between your pure interpreter and your IO-based interpreter in terms of whether the key-value store effects that m1 performed before it failed are visible when interpreting m2? If so, why?
Yes, there is a significant difference between the pure and IO-based interpreters regarding the visibility of key-value store effects performed by m1.

Pure Interpreter:

- In a pure interpreter, the state is generally immutable. When an operation (like TryCatchOp m1 m2) is performed, if m1 fails, it does not alter the state of the key-value

store. Thus, any effects (like updates or inserts in the key-value store) performed by m1 before it fails are **not visible** when interpreting m2.
- This is because the pure interpreter would maintain a separate state for the execution of m1, and if m1 fails, it simply discards the changes made to that state.

IO-Based Interpreter:

- In contrast, an IO-based interpreter may allow side effects, meaning that any operations performed in m1 could affect the actual state of the key-value store.
- If m1 fails after making changes, those changes might still persist in the key-value store, depending on how the transaction and error handling are implemented. So, when interpreting m2, you might see the effects of the operations in m1 if they were executed before the failure.

(b) Suppose you've implemented your interpreters such that the key-value store effects that m1 performed before it failed are always visible when interpreting m2. Without changing the interpreters, is it possible to have different behavior where the key-value store effects in m1 are invisible in m2? If so, how? If not, why not?

Yes, it is possible to have different behavior. By implementing rollback mechanisms or isolation contexts, I can ensure that changes in m1 do not affect m2, even if my current implementation allows those effects to be visible.

Possible Ways to Achieve Different Behavior:

- **Manual Rollback Mechanism:** I could implement a rollback mechanism that allows the interpreter to revert the state after a failure. For instance, if m1 fails, I could maintain a snapshot of the state before executing m1, allowing m2 to be executed in the original state.
- **Isolation:** By creating an isolated context for m1 where all effects are local, I can ensure that changes are not reflected in m2. This is similar to how databases handle transactions, where a transaction can be committed or rolled back based on success or failure.

2. Why does the computation payload in the TransactionOp (EvalM ()) a constructor return a () value? Do any other return types make sense? Justify your answer.

The computation payload in the TransactionOp (EvalM ()) a constructor returns a () value because the primary purpose of a transaction operation is to execute a series of effects (like key-value store updates) without needing to produce a meaningful result. The () type indicates that the transaction's success or failure does not depend on a return value, but rather on the side effects of the operations performed within the transaction.

Justification:

1. **Side Effects Over Results**: In a transactional context, the focus is on the correctness and atomicity of the operations rather than the value produced. The transaction's effects (such

as updates to the key-value store) are what matter, and these can be verified or observed through subsequent operations rather than through the transaction's return value.

2. **Consistency and Rollback**: The () return type aligns with the idea that if the transaction fails, no changes should be made to the underlying state. Thus, there's no meaningful value to return—only the acknowledgment that an attempt to perform some operations was made.
3. **Alternative Return Types**: While returning other types could make sense if you needed to convey additional information (like the outcome of specific operations), it complicates the transaction semantics. For instance, if a transaction returned an error type or a result type, it might imply that the transaction can produce a value, which conflicts with the notion of atomicity. Therefore, while technically possible, other return types would not align with the intended design and use of transactions in this context.