This project focused on implementing a subset of the heap checker portion for Gallifrey types. The implementation for the project can be found at: `https://github.com/fishy15/minifrey-types`. I worked alone on this project.

1 Problem Description

With concurrent programs, there is a need to share data between different threads. However, we can run into race conditions between threads running in parallel trying to modify the same data structures. We can add locks, but we run the risk of creating deadlocks.

Instead of this, another model used by languages such as Erlang is message passing. However, if we wanted to send a complex data structure from one thread to another, this would require a complete deep copy. Instead, a pointer can be sent from one thread to another, and the original thread should lose access of the data structure. Languages such as Go allow for this option.

If we do send a pointer, there is a risk that the original thread still contains an alias to the sent object. At runtime, this could lead to undefined behavior or race conditions. What the region checker in the project aims to do add allow the programmer to add certain annotations and restrictions to the source code to check at compile time if the threads attempts to access any such aliases.

Languages such as Rust use an ownership model, where every object has a unique owner associated with it. This works for certain data structures, but other data structures such graphs with cycles in it can be more difficult to represent. Instead of enforcing every object must have a unique owner, only certain objects, known as *isolating* references. This allows for greater flexibility while still keeping static guarantees about the program.

2 Solution Overview

2.1 Isolating and Regular References

During the execution of a program, each thread will have an associated *reservation*, which is the portion of the heap that belongs to the thread. With sends and receives, reservations shrink and grow respectively. To model this at compile time, the checker associates each reference with a *region*. If two pointers are of the same type and in the same region, they potentially could be aliases of each other. The programmer does not need to annotate which region the object a pointer points to. Instead, the compiler is able to compute which region an object belongs to using other annotations.

In addition, the programmer has to annotate if variables are isolated references or regular references. We can construct a directed graph of objects, where an edge exists from an object to each object it contains a reference to. Isolated references are supposed to *globally dominate* their reference graph. Specifically, suppose that x is an isolating reference, and y is some reference reachable from x. For x to satisfy the global domination invariant, every path from the root of the reference graph to y must go through x. Regular references do not need to satisfy any requirements.

Suppose some object contains an isolating reference. The checker marks that the object and the isolating reference are in separate regions. If the isolating reference was in the same region as the original object, that would imply that they are potential aliases, which would break the global domination invariant. On the other hand, because there are no restrictions with regular references, the reference could potentially be an alias, so the object and a regular reference it contains must be in the same region.

For instance, consider a binary search tree. In this case, the references from a node to its children would both be isolating references. There is a unique path from the root to any node in the tree, so the global domination property would be satisfied. Each of these nodes would be in their own region.

On the other hand, consider a general graph node, where the root of the object graph is some arbitrary node. If the graph is a tree, then there are unique path from the root of the object graph to everything, so they could be isolating references. However, other graphs such as ones with cycle would break the global

domination invariant. Therefore, these have to be regular references. These also all must be in the same region since a chain of references can lead back to the original node.

2.2 Tracking References

In addition to isolating and regular references, there are also tracking references. In certain cases, we may want to process an object inside the data structure. For example, suppose we implement finding a node in a binary search tree by looping until we find the node, storing a reference to the current node. However, creating another reference for this processing would violate the global domination invariant since there would be a path that goes through the new alias instead of through the original data structure.

To fix this, the checker actually enforces a *tempered domination invariant*. Specifically, some pointers are explicitly marked as tracking pointers. Each region can have at most one tracking reference, even if the references are different types. The global domination invariant is enforced for every region that does not have a tracking reference. Because each region has at most one tracking reference, this prevents creating multiple tracking references to the same object. This is different from the Gallifrey type system, which is able to infer which references should be tracking references.

2.3 Sending Messages

When we send a reference, every single reference to the same region is also invalidated. This is because within a region, any two objects could potentially be reachable or not. In addition, every region reachable from the current region is also invalidated.

In other words, the graph of regions can be thought of as a tree, with isolating references representing the edges in the tree. Every single region in the subtree rooted at a region is reachable from that region. Therefore, every reference within the subtree will be marked as invalid.

If the user wants to keep some portion of the region instead of the checker marking it as invalid, the user needs to mark certain references as isolated references. All the regions higher in the region tree will not be marked invalid, so adding more isolated reference annotations reduces the number of variables marked invalid. The checker will make sure that the tempered domination invariant is still maintained.

2.4 Functions

With this implementation of the checker, all parameters are required to be in regions unreachable from each other. When the program returns, the regions the parameters originally were in are required to be the regions they end up in, and they must still be unreachable from each other. In addition, the value returned must be in a new region.

This is done because we want to preserve the same shape of the graph before and after calling the function. For example, suppose we call a function with two parameters x and y, where x and y are originally not reachable from one another. If we were allowed to change this inside of the function, then this reachability constraint would be changed after we return from the function as well. Handling this would require each function call to also process how the call function affects the region context at that specific call, which may not be feasible in the general case. Instead of allowing for the free variation, the checker enforces that the function does not modify the region context.

3 Implementation

The model programming language simulates assignments, sending, receiving, and function calls. The project focused on these expressions since they represent the different region manipulations can be done. The checker, implemented in Haskell, checks the regions of each program based on the type and reference type annotations given by the user.

There are 9 types of expressions in the model programming language:

- `New` — create an object of a certain type
- `AccessVar` — access the value of a local variable
- `AccessField` — access the value of a given field of a local variable
- `AssignVar` — set the value of a local variable
- `AssignField` — set the value of a given field of a local variable
- `Seq` — execute one expression, execute another expression, and return the value of the second expression
- `FuncCall` — call a function with the given expressions as inputs
- `Send` — send the output of an expression away from the thread
- `Receive` — receive an object of a certain type from another thread

To type check a function, the program first assumes that each function parameter is from a separate region, and thus each parameter can be added to the set of local variables as isolating references. Expressions such as `New`, `FuncCall`, and `Receive` create a new expression in a new region, so they return an isolating reference in a new region. Other expressions such as `AccessVar` and `AccessField` search for the variable in the current context and return the region it belongs to.

The behavior on how `AssignVar` and `AssignField` change on if the reference type is `Iso`/`Tracking`, or `Regular`. For `Iso` and `Tracking` references, they can always point to a new region, so there are no restrictions on which region it can point to. On the other hand, if a `Regular` reference could be reassigned to any region, it would mean that there references in the region of the struct and references in the new region of the regular reference could potentially now alias each other, forcing both regions to be merged together. For this reason, regular references can only be assigned to other regular references in the same region.

`FuncCall` has to make sure the assumptions at the beginning of a function call are correct. The initial assumption is that every parameter is in a distinct region unreachable from the regions of the other parameters, and the checker makes sure this is true for every expression in the function parameter list. The final assumptions are that each parameter remains in its own region, which are still unreachable from each other, and the return value is also in an unreachable reason. The checker can assume that this is true and return a new variable in a new context.

Inside our program state, the checker stores the set of regions which have been sent so far. As mentioned earlier, the set of regions which are reachable from a sent region are the regions which are invalidated over time. Instead of finding every invalid reference in the current context and marking it as invalid, the checker simply adds the sent region to the set and checks if a program tries to use a region that is reachable from a sent region.

Finally, after computing the return type of a function, the checker needs to make sure the final assumptions about a function are correct. Specifically, the checker loops through all the parameters and make sure they still have valid references in the same original regions. In addition, we make sure the return value is a valid reference in a new region. Because we only check these invariants at the beginning and end of a function, this allows some flexibility within the actual function while still maintaining region safety.

4 Results

The checker is able to check for region correctness with various program structures. As an example, consider the following two programs:

```
buildTree() {
    root = new Tree;
    left = recv<Tree>();
    right = recv<Tree>();
    root.left = left;
    root.right = right;
    send(root)
}
```

```
buildAndFindMin() {
    root = new Tree;
    left = recv<Tree>();
    right = recv<Tree>();
    root.left = left;
    root.right = right;
    send(root);
    findMin(root.left)
}
```

The left function does not use-after-send or break the tempered domination invariant, so it should be valid. The program is represented as an ADT in Haskell in the `buildTree` test case, and the checker correctly marks that it is a success.

On the other hand, the second function attempts to use after sending a variable. The program is represented as an ADT in Haskell in the `buildAndFindMin` test case. The checker is able to correctly determine that we are accessing a reference that is reachable from a sent reference, so it errors while checking. Other examples can be found in `src/run.hs`.

5 Future Work

5.1 Improving the Checker

In the future, more features could be added to make the checker more robust. For example, more thorough type checking can be added. Currently, the checker assumes that the types for every expression matches the intended type.

Another feature that could be added is inferring reference annotations. The user still would have to annotate the values of a struct, but function variables could be inferred automatically. Certain expressions always return isolating references or tracking references, and so the variable they are assigned to can either be assigned isolating or tracking based on whichever one is not taken. Other expressions always return regular references, and so they can always be assigned to regular references.

Finally, support for other types of expressions can be added. This project focused on assigning and referring to references, but a full language would require constants as well. These are copied by value and sent by value, so it does not directly apply to the regional analysis in the project, but it would be important in expanding this checker to one for a full language.

5.2 Expanding Regional Analysis

One way the checker can be expanded is by adding support for conditional statements. In situations such as simply-typed lambda calculus, we simply have to check if the types in both branches match. However, in this checker, we would also need to check if the region context between both branches can be made to match through some set of transformations. In the Gallifrey type system, a set of rules called *virtual transformations* enumerate these transformations, which are changes to the region context independent of syntax. Some of these transformations include creating a fresh region or merging two regions together.

For one instance when these transformations are important, consider the `if disconnected` expression, which is used to check if two values in the same region are disconnected at runtime. The then branch will explicitly mark the regions as separate in the region context, while the else branch will keep them as the same. In order to unify these branches again, we need to use the merge virtual transformation mentioned earlier to reunite the separate regions.

Another way the syntax can be expanded is by allowing for more expressive function types. We earlier made the assumption that every parameter starts in its own region and ends up in its own region. If we could fully specify how the region context changes as a result of calling the function, we could allow some more flexibility

in the types of functions that we allow. The Gallifrey type system uses mentions the `consumes` keyword, which indicates to the type system that a parameter's region becomes invalid after calling the function.

Finding the right annotation for functions may be difficult in certain cases as well. Consider a binary search tree, where each node has isolating references to its children. The requirement currently in place is that the returned value has to be in its own region unreachable from the parameters. This works if we want to delete a value from the binary search tree since deleting a value separates it from the regions of the original tree.

However, if we want to return an alias without deleting the value, we wouldn't be able to under the current region type system implemented in this project. We could potentially add an annotation stating that the returned reference lives in the region of some node in the binary search tree, but there is no way of knowing in advance which region the return value belongs to. More work can be done on making situations like these easier to express.

5.3 Simpler Runtime Checks

Some situations with the Gallifrey type system incur a runtime cost to satisfy the type checker. One example the paper mentions are circular doubly-linked lists. Part of the implementation checks if two isolating references from the same region alias each other. The type system itself cannot determine this at compile time, so a runtime check is necessary.

If the object graph grows large, then this runtime check may be expensive, especially if the guideline the paper mentions of keeping regions small is not followed. In certain cases, it may be useful to have a faster check than `if disconnected` that can still satisfy the type system. One possibility for the circular doubly-linked list is checking if the owner of the isolating reference equals the owner of the other reference. In certain cases, this may be faster, but it requires the programmer to be more careful to deal with cases of multiple objects having isolating references to the same object.

6 References

"A Flexible Type System for Fearless Concurrency" — `https://www.cs.cornell.edu/andru/papers/gallifrey-types/gallifrey-types.pdf`

"An accessible introduction to type theory and implementing a type-checker" — `https://mukulrathi.com/create-your-own-programming-language/intro-to-type-checking/`

"Dominance: Modeling Heap Structures with Sharing" — `https://www.cs.unm.edu/~treport/tr/07-08/heap-structure.pdf`

"Message-passing concurrency in Erlang" — `https://www.cse.chalmers.se/edu/year/2016/course/TDA383_LP3/files/lectures/Lecture08-message-passing.pdf`

"Types for Immutability and Aliasing Control" — `http://ceur-ws.org/Vol-1720/full5.pdf`