Report on the f-RGN Programming Language[*]

Xuanrui Qi
Department of Computer Science
Tufts University
`xuanrui.qi@tufts.edu`

Supervisor: Dr. Mark A. Sheldon

May 15, 2017

Abstract

`f-RGN` is a programming language based on the calculus $\mathbf{F}^{\mathbf{RGN}}$, described in Fluet, Morrisett and Ahmed's paper "Linear Regions Are All You Need" [1]. It allows for compile-time type inference and provides a small number of basic, literal types. This report provides the static and dynamic semantical rules and a brief overview of the language.

1 Introduction

`f-RGN` is a functional, statically typed programming language designed and presented by this author in this report, based on the work in the calculus $\mathbf{F}^{\mathbf{RGN}}$ presented in Fluet, Morrisett and Ahmed's work[1]. However, $\mathbf{F}^{\mathbf{RGN}}$ is not efficiently and succintly implementable due to it having a large number of constructs that are purely theoretical, and the requirement for explicit type signatures. Therefore, we have reduced the complexity of $\mathbf{F}^{\mathbf{RGN}}$ by removing some monadic operations and limiting polymorphic qualifiers to be compatible with Hindley-Milner style type inference. Moreover, since `f-RGN` is meant to be practical, it adds some common concrete types to $\mathbf{F}^{\mathbf{RGN}}$.

2 Goals

In creating this project, we intend to explore the static techniques and guarantees for memory safety and the prevention of leaks. We surveyed a number of languages, including ATS, Rust, Cyclone, and the calculus $\mathbf{F}^{\mathbf{RGN}}$, as well as a number of conceptual tools, including linear types, separation logic, and region

[*]This is a report completed for Comp 193: Directed Study at Tufts University.

& effect systems. We decided to base our language on $\mathbf{F}^{\mathbf{RGN}}$ because of its simplicity and functional purity.

3 Motivation

It is hard to write programs that are memory safe without leaking any memory. Traditionally, programming systems have used dynamic methods at runtime, such as reference counting or garbage collection, to enforce those guarantees; however, garbage collection algorithms can pose a severe challenge to performance. The use of regions, which could be reasoned about at compile-time, allows programmers to easily write programs without memory access violations or memory leaks.

In the FX-91 programming language, regions were used as a form of memory access guarantee, but it did not go as far as preventing memory leaks [2]. Tofte and Talpin presented a programming language implemented using a stack of regions [3]; although it did not fully guarantee the lack of memory leaks, it is experimentally shown that the memory requirements are much reduced even without garbage collection [3].

4 Example Programs

The following function allocates an integer on a given stack and returns a pointer to the integer. It has the type `hnd` $\varsigma \rightarrow$ `int` $\rightarrow$ `ref` ς `int`, where `s0` is the index of the newly created stack. Note that `h` could not change after `run-rgn`, as no stateful transformations could escape `run-rgn`; therefore, `run-rgn` cannot leak any memory as well:

```
(lambda h (lambda x (run-rgn (new h 3))))
```

It is also possible to access newer regions when we are in an older stack. We assume `h` is a handle to a stack:

```
(let-rgn h
  (write 3
    (run-rgn (new h 10))))
```

In the code snippet above, we allocated an integer 10 ”above” our current region in the same stack, wrote an integer 3 into the region, and destroys the region. This design of `let-rgn` guarantees the absence of memory leaks.

5 Syntax

The concrete syntax of `f-RGN` is defined as following, in Extended Backus-Naur form:

$\langle program \rangle$ ::= [$\langle expression \rangle$ | $\langle definition \rangle$]

```
⟨expression⟩  ::= ⟨ident⟩
               |  ⟨literal⟩
               |  (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
               |  (lambda ⟨ident⟩ ⟨expression⟩)
               |  (⟨expression⟩ {⟨expression⟩})
               |  ()
               |  (pair ⟨expression⟩ ⟨expression⟩)
               |  (unpack (⟨ident⟩ ⟨ident⟩) ⟨expression⟩ ⟨expression⟩)

⟨literal⟩     ::= ⟨string⟩ | ⟨numeric⟩ | #t | #f

⟨string⟩      ::= ''⟨character⟩''

⟨definition⟩  ::= (define ⟨ident⟩ ⟨expression⟩)
               |  (defun ⟨ident⟩ ⟨ident⟩ ⟨expression⟩)

⟨ident⟩       ::= ⟨alpha⟩, { ⟨alpha⟩ | ⟨numeric⟩ | -_?! }

⟨alpha⟩       ::= { a-z A-Z }

⟨numeric⟩     ::= { 0-9 }
```

Logically, all functions have only one parameter, as in the λ-calculus. However, syntactic sugar is provided for applying curried functions; `(f x y z)` is equivalent to `(((f x) y) z)`. Similarly, the only definition form involves defining a variable with a value; `defun` is provided as syntactic sugar for `define`. `(defun f x e)` is equivalent to `(define f (lambda x e))`.

All definitions must appear at the top level and all variables are statically typed; that is, no evaluation of any given expression could introduce new variables or eliminate variables from any environment, nor could any evaluation alter the environments of types and/or kinds.

Monadic region operations are not given special treatment in the concrete syntax; they are considered built-in functions of `f-RGN`. However, the identifiers `return`, `then`, `new`, `read`, `write`, `run-rgn`, `let-rgn`, and `get-hnd` are considered reserved identifiers and should not be used as names.

Stack indices are explicit in `f-RGN`. However, they are generated automatically during type inference, not by the programmer. The prototype type inferencer use the names "s#" for stack indices, where # is a natural number. Therefore, programmers should avoid these names when naming variables.

Currently, `f-RGN` has no support for defining and/or expanding macros. The only way to add new expression forms and syntax sugar is by altering the implementation.

6 The Region Monad

There are 9 monadic region operations, `return`, `>>=`, `new`, `read`, `write`, `run-rgn`, `let-rgn`, `new-stack`, and `get-hnd`. They have the following types:

$\texttt{return} : \forall\varsigma.\ \forall\alpha.\ \alpha \to \texttt{rgn}\ \varsigma\ \alpha$
$\texttt{>>=}\ (\texttt{then}) : \forall\varsigma.\ \forall\alpha,\beta.\ \texttt{rgn}\ \varsigma\ \alpha \to (\alpha \to \texttt{rgn}\ \varsigma\ \beta) \to \texttt{rgn}\ \varsigma\ \beta$
$\texttt{new} : \forall\varsigma.\ \forall\alpha.\ \texttt{hnd}\ \varsigma \to \alpha \to \texttt{rgn}\ \varsigma\ (\texttt{ref}\ \varsigma\ \alpha)$
$\texttt{read} : \forall\varsigma.\ \forall\alpha.\ \texttt{ref}\ \varsigma\ \alpha \to \texttt{rgn}\ \varsigma\ \alpha$
$\texttt{write} : \forall\varsigma.\ \forall\alpha.\ \texttt{ref}\ \varsigma\ \alpha \to \alpha \to \texttt{rgn}\ \varsigma\ \texttt{unit}$
$\texttt{run-rgn} : \forall\varsigma.\ \forall\alpha.\ \texttt{rgn}\ \varsigma\ \alpha \to \alpha$
$\texttt{let-rgn} : \forall\varsigma_1,\varsigma_2.\ \forall\alpha.\ \texttt{hnd}\ \varsigma_2 \to \texttt{rgn}\ \varsigma_2\ \alpha \to \texttt{rgn}\ \varsigma_1\ \alpha$
$\texttt{new-stack} : \forall\varsigma.\ \texttt{hnd}\ \varsigma$
$\texttt{get-hnd} : \forall\varsigma.\ \forall\alpha.\ \texttt{rgn}\ \varsigma\ \alpha \to \texttt{hnd}\ \varsigma$

The `new-stack` and `get-hnd` operations are not defined in [1], and are our own additions to the language. `new-stack` allocates an empty stack of regions and returns a handle to it, while `get-hnd` takes a region computation and returns a handle to the region encapuslated in the computation.

7 Typing Rules

The inference rules for `f-RGN` follows that of a Hindley-Milner typed lambda calculus in general, with a few additions. Hereafter, we use $\Delta : \epsilon \to \kappa$ to represent an environment of kinds, where ϵ is either a valid type or a stack index, and κ is a valid kind; we use $\Gamma : x \to \tau$ to represent an environment of types, where x is a valid identifier in `f-RGN` and τ a valid type.

Note that *all well-typed expressions have a type of the kind* * by design. This is the reason why `get-hnd` takes a `rgn` but never directly a stack - stack indices cannot be exposed at the expression level! If they do, type inference will be much harder and possibly impossible.

$\kappa ::= \texttt{STACK} \mid *$
$\text{BaseType} ::= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{string}$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\textsc{Var}\quad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\textsc{IfElse}\quad \frac{\Delta; \Gamma \vdash e_c : \texttt{bool} \qquad \Delta; \Gamma \vdash e_t : \tau \qquad \Delta; \Gamma \vdash e_f : \tau}{\Delta; \Gamma \vdash (\texttt{if}\ e_c\ e_t\ e_f) : \tau}$$

$$\textsc{Lambda}\quad \frac{\Delta; \Gamma, x : \tau \vdash e : \tau'}{\Delta, \Gamma \vdash (\texttt{lambda}\ x\ e) : \tau \to \tau'}$$

$$\textsc{App}\quad \frac{\Delta; \Gamma \vdash e_1 : \tau \to \tau' \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash (e_1\ e_2) : \tau'}$$

$$\textsc{Unit}\quad \frac{}{\Delta; \Gamma \vdash () : \texttt{unit}}$$

$$\textsc{Pair}\quad \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (\texttt{pair}\ e_1\ e_2) : \tau_1 \times \tau_2}$$

$$\textsc{Unpack}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e_p : \tau_1 \times \tau_2 \\ \Delta; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau\end{array}}{\Delta; \Gamma \vdash \texttt{(let (}x_1\ x_2\texttt{)}\ e_p\ e\texttt{)} : \tau}$$

8 Type Inference and Its Implementation

Unlike in the calculus $\mathbf{F^{RGN}}$, there is no need to explicitly specify types in the `f-RGN` programming language. Type inference, based on Algorithm R, as described in Turbak, Gifford with Sheldon [4], is possible.

The type inference rules of `f-RGN` are as following. The function inf is the type inference function, taking an expression, a type environment and a kind environment and returning a pair of an inferred type and a type constaint.

InfA $::= \tau \mid \texttt{FAIL}$ is the answer to an inference
$inf : e \to \Gamma \to \Delta \to (\tau \times \text{TypeConstraintSet})$
$\tau_1 \doteq \tau_2$ is a constraint that equates τ_1 and τ_2

UnitInf

$$inf[\![()]\!]\ \Gamma\ \Delta = \langle \texttt{unit}, \{\} \rangle$$

LiteralInf

$$\frac{m \text{ is a literal of type } \tau \qquad \tau \in \{\texttt{int}, \texttt{string}, \texttt{bool}\}}{inf[\![m]\!]\ \Gamma\ \Delta = \langle \tau, \{\} \rangle}$$

VarInf

$$\frac{x \in \mathrm{dom}(\Gamma)}{inf[\![x]\!]\ \Gamma\ \Delta = \langle \Gamma(x), \{\} \rangle}$$

IfElseInf

$$\frac{\begin{array}{c} inf[\![e_c]\!]\ \Gamma\ \Delta = \langle \tau_c, TCS_c \rangle \\ inf[\![e_t]\!]\ \Gamma\ \Delta = \langle \tau_t, TCS_t \rangle \\ inf[\![e_f]\!]\ \Gamma\ \Delta = \langle \tau_f, TCS_f \rangle \end{array}}{inf[\![(\texttt{if}\ e_c\ e_t\ e_f)]\!]\ \Gamma\ \Delta = \langle \tau_t, TCS_c \cup TCS_t \cup TCS_f \cup \{\tau_c \doteq \texttt{bool}, \tau_t \doteq \tau_f\} \rangle}$$

LambdaInf

$$\frac{\begin{array}{c} inf[\![e]\!]\ (\Gamma, x : \tau_x)\ (\Delta, x : *) = \langle \tau_e, TCS_e \rangle \\ \tau_x \text{ is a fresh type variable} \end{array}}{inf[\![(\texttt{lambda}\ x\ e)]\!]\ \Gamma\ \Delta = \langle \tau_x \to \tau_e, TCS_e \rangle}$$

ApplyInf

$$\frac{\begin{array}{c} inf[\![e_1]\!]\ \Gamma\ \Delta = \langle \tau_f, TCS_1 \rangle \qquad inf[\![e_2]\!]\ \Gamma\ \Delta = \langle \tau, TCS_2 \rangle \\ \tau' \text{ is a fresh type variable} \end{array}}{inf[\![(e_1\ e_2)]\!]\ \Gamma\ \Delta = \langle \tau', TCS_1 \cup TCS_2 \cup \{\tau_f \doteq \tau \to \tau'\} \rangle}$$

PairInf

$$\frac{\begin{array}{c} inf[\![e_1]\!]\ \Gamma\ \Delta = \langle \tau_1, TCS_1 \rangle \\ inf[\![e_2]\!]\ \Gamma\ \Delta = \langle \tau_2, TCS_2 \rangle \end{array}}{inf[\![(\texttt{pair}\ e_1\ e_2)]\!]\ \Gamma = \langle \tau_1 \times \tau_2, TCS_1 \cup TCS_2 \rangle}$$

UnpackInf

$$\frac{\begin{array}{c} inf[\![e_p]\!]\ \Gamma\ \Delta = \langle \tau, TCS_1 \rangle \\ inf[\![e]\!]\ (\Gamma, x_1 : \tau_1, x_2 : \tau_2)\ (\Delta, x_1 : *, x_2 : *) = \langle \tau', TCS_2 \rangle \\ \tau_1, \tau_2 \text{ are fresh type variables} \end{array}}{inf[\![(\texttt{let}\ (x_1\ x_2)\ e_p\ e)]\!] = \langle \tau', TCS_1 \cup TCS_2 \cup \{\tau \doteq \tau_1 \times \tau_2\} \rangle}$$

ReturnInf

$$\frac{inf[\![e]\!]\ \Gamma\ \Delta = \langle \tau, TCS \rangle \qquad \varsigma \text{ is a fresh stack index}}{inf[\![(\texttt{return}\ e)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma\ \tau, TCS \rangle}$$

$$\textsc{>>=Inf}\quad \frac{\begin{array}{c} inf[\![m]\!]\ \Gamma\ \Delta = \langle \tau_1, TCS_1\rangle \qquad inf[\![f]\!]\ \Gamma\ \Delta = \langle \tau_2, TCS_2\rangle \\ \varsigma \text{ is a fresh stack index} \qquad \alpha, \beta \text{ are fresh type variables}\end{array}}{inf[\![(>>=\ m\ f)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma\ \beta, TCS_1 \cup TCS_2 \cup \{\tau_1 \doteq \texttt{rgn}\ \varsigma\ \alpha, \tau_2 \doteq \alpha \to \texttt{rgn}\ \varsigma\ \beta\}\rangle}$$

$$\textsc{NewInf}\quad \frac{\begin{array}{c} inf[\![h]\!]\ \Gamma\ \Delta = \langle \tau_1, TCS_1\rangle \\ inf[\![e]\!]\ \Gamma\ \Delta = \langle \tau_2, TCS_2\rangle \qquad \varsigma \text{ is a fresh stack index}\end{array}}{inf[\![(\texttt{new}\ h\ e)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma\ (\texttt{ref}\ \varsigma\ \tau_2), TCS_1 \cup TCS_2 \cup \{\tau_1 \doteq \texttt{hnd}\ \varsigma\}\rangle}$$

$$\textsc{ReadInf}\quad \frac{\begin{array}{c} inf[\![p]\!]\ \Gamma\ \Delta = \langle \tau, TCS\rangle \\ \varsigma \text{ is a fresh stack index} \qquad \alpha \text{ is a fresh type variable}\end{array}}{inf[\![(\texttt{read}\ p)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma\ \alpha, TCS \cup \{\tau \doteq \texttt{ref}\ \varsigma\ \alpha\}\rangle}$$

$$\textsc{WriteInf}\quad \frac{\begin{array}{c} inf[\![p]\!]\ \Gamma\ \Delta = \langle \tau, TCS\rangle \qquad inf[\![e]\!]\ \Gamma\ \Delta = \langle \tau', TCS'\rangle \\ \varsigma \text{ is a fresh stack index}\end{array}}{inf[\![(\texttt{write}\ p\ e)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma\ \texttt{unit}, TCS \cup TCS' \cup \{\tau \doteq \texttt{ref}\ \varsigma\ \tau'\}\rangle}$$

$$\textsc{RunRgnInf}\quad \frac{\begin{array}{c} inf[\![e]\!]\ \Gamma\ \Delta = \langle \tau, TCS\rangle \\ \varsigma \text{ is a fresh stack index} \qquad \alpha \text{ is a fresh type variable}\end{array}}{inf[\![(\texttt{run}-\texttt{rgn}\ e)]\!]\ \Gamma\ \Delta = \langle \alpha, TCS \cup \{\tau \doteq \texttt{rgn}\ \varsigma\ \alpha\}\rangle}$$

$$\textsc{LetRgnInf}\quad \frac{\begin{array}{c} inf[\![h]\!]\ \Gamma\ \Delta = \langle \tau, TCS\rangle \qquad inf[\![m]\!]\ \Gamma\ \Delta = \langle \tau', TCS'\rangle \\ \varsigma_1, \varsigma_2 \text{ are fresh stack indices} \qquad \alpha \text{ is a fresh type variable}\end{array}}{inf[\![(\texttt{let}-\texttt{rgn}\ h\ m)]\!]\ \Gamma\ \Delta = \langle \texttt{rgn}\ \varsigma_1\ \alpha, TCS \cup TCS' \cup \{\tau \doteq \texttt{hnd}\ \varsigma_2, \tau' \doteq \texttt{rgn}\ \varsigma_2\ \alpha\}\rangle}$$

$$\textsc{NewStackInf}\quad \frac{\varsigma \text{ is a fresh stack index}}{inf[\![\texttt{new}-\texttt{stack}]\!]\ \Gamma\ \Delta = \langle \texttt{hnd}\ \varsigma, \{\}\rangle}$$

$$\textsc{GetHndInf}\quad \frac{\begin{array}{c} \mathrm{inf}[\![r]\!]\ \Gamma\ \Delta = \langle \tau, TCS\rangle \\ \varsigma \text{ is a fresh stack index} \qquad \alpha \text{ is a fresh type variable}\end{array}}{\mathrm{inf}[\![(\texttt{get}-\texttt{hnd}\ r)]\!]\ \Gamma\ \Delta = \langle \texttt{hnd}\ \varsigma\rangle, TCS \cup \{\tau \doteq \texttt{rgn}\ \varsigma\ \alpha\}}$$

The type inference algorithm for `f-RGN` is implemented in Haskell as following. The following Haskell packages are required: External Haskell pacakges:

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}

import qualified Data.Map.Strict as Map
import qualified Data.Set as Set

import Control.Monad.Except
import Control.Monad.Reader
import Control.Monad.State
```

The data structures associated with type inference are defined as following:

```
type TyEnv = Env Type
type KindEnv = Env Kind

type TySubst = Map.Map Id TyTerm
type TyConsSet = Map.Map Type Type
```

As generating fresh type variables require state, we wrap inf in a state monad:

```
inf :: Expr → TyEnv → KindEnv → Inference (Type, TyConsSet)
inf Empty _ _ = return (Unit, Map.empty)
inf (Literal m) _ _ = case m of
                              I _ → return (Integer, Map.empty)
                              B _ → return (Bool, Map.empty)
                              S _ → return (String, Map.empty)
inf (Variable x) tEnv _ = case Map.lookup x tEnv of
                                  Nothing → throwError $ "Unbound␣variable" ++
x
                                  Just ty → return (ty, Map.empty)
inf (IfElse ec et ef) tEnv kEnv =
    do
        (tauC, tcsC) ← inf ec tEnv kEnv
        (tauT, tcsT) ← inf et tEnv kEnv
        (tauF, tcsF) ← inf ef tEnv kEnv
        return (tauT, tcsC `Map.union` tcsT `Map.union` tcsF `Map.union`
                         (Map.singleton tauC Bool) `Map.union`
                         (Map.singleton tauT tauF))
inf (Lambda x e) tEnv kEnv =
    do
        tauX ← freshTyVar "t"
        let tEnv0 = Map.delete x tEnv
            tEnv' = Map.union tEnv0 (Map.singleton x tauX)
            kEnv' = Map.union kEnv (Map.singleton x Star)
        (tauE, tcsE) ← inf e tEnv' kEnv'
        return (Arrow tauX tauE, tcsE)
inf (Apply e1 e2) tEnv kEnv =
    do
        tau' ← freshTyVar "t"
        (tauF, tcs1) ← inf e1 tEnv kEnv
        (tau,  tcs2) ← inf e2 tEnv kEnv
        return (tau', tcs1 `Map.union` tcs2 `Map.union` (Map.singleton tauF (Arrow tau tau')))
inf (Pair e1 e2) tEnv kEnv =
    do
        (tau1, tcs1) ← inf e1 tEnv kEnv
        (tau2, tcs2) ← inf e2 tEnv kEnv
        return (Product tau1 tau2, tcs1 `Map.union` tcs2)
inf (Unpack (x1, x2) ep e) tEnv kEnv =
    do
        tau1 ← freshTyVar "t"
        tau2 ← freshTyVar "t"
        (tau, tcs1) ← inf ep tEnv kEnv
        let tEnv' = Map.insert x1 tau1 (Map.insert x2 tau2 (Map.delete x1
                                                (Map.delete x2 tEnv)))
            kEnv' = Map.insert x1 Star (Map.insert x2 Star kEnv)
        (tau', tcs2) ← inf e tEnv' kEnv'
        return (tau', tcs1 `Map.union` tcs2 `Map.union` (Map.singleton tau (Product tau1 tau2)))
inf (Return e) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        (tau, tcs) ← inf e tEnv kEnv
        return (Region s tau, tcs)
inf (Then m f) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        alpha ← freshTyVar "t"
        beta  ← freshTyVar "t"
        (tau1, tcs1) ← inf m tEnv kEnv
        (tau2, tcs2) ← inf f tEnv kEnv
        return (Region s beta, tcs1 `Map.union` tcs2 `Map.union`
                Map.fromList [(tau1, Region s alpha), (tau2, Arrow alpha (Region s beta))])
inf (New h e) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        (tau1, tcs1) ← inf h tEnv kEnv
        (tau2, tcs2) ← inf e tEnv kEnv
        return (Region s (Reference s tau2), tcs1 `Map.union` tcs2 `Map.union`
                (Map.singleton tau1 (Handle s)))
inf (Read p) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        alpha ← freshTyVar "t"
        (tau, tcs) ← inf p tEnv kEnv
        return (Region s alpha, tcs `Map.union` (Map.singleton tau (Reference s alpha)))
inf (Write p e) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        (tau, tcs) ← inf p tEnv kEnv
        (tau', tcs') ← inf e tEnv kEnv
        return (Region s Unit, tcs `Map.union` tcs' `Map.union`
                (Map.singleton tau (Reference s tau')))
inf (RunRgn e) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        alpha ← freshTyVar "t"
        (tau, tcs) ← inf e tEnv kEnv
        return (alpha, tcs `Map.union` (Map.singleton tau (Region s alpha)))
inf (LetRgn h m) tEnv kEnv =
    do
        s1 ← freshStkIdx "s"
        s2 ← freshStkIdx "s"
        alpha ← freshTyVar "t"
        (tau, tcs) ← inf h tEnv kEnv
        (tau', tcs') ← inf m tEnv kEnv
        return (Region s1 alpha, tcs `Map.union` tcs' `Map.union`
                (Map.fromList [(tau, Handle s2), (tau', Region s2 alpha)]))
inf (NewStk) _ _ =
    do
        s ← freshStkIdx "s"
        return (Handle s, Map.empty)
inf (GetHnd r) tEnv kEnv =
    do
        s ← freshStkIdx "s"
        alpha ← freshTyVar "t"
        (tau, tcs) ← inf r tEnv kEnv
        return (Handle s, tcs `Map.union` (Map.singleton tau (Region s alpha)))
```

We may then attempt to unify all the typing constraints collected in the *inf* function and infer the types for all variables:

```
data UnifySoln = Solution TySubst | Fail
    deriving (Show)

unifyLoop :: TyConsSet → TySubst → UnifySoln
unifyLoop tcs subst =
    let tcs' = Map.toList tcs
    in  case tcs' of
            [] → Solution subst
            ((t, t'):rest) →
                if eqType t t'
                then unifyLoop (Map.fromList rest) subst
                else case (t, t') of
                    (Arrow t1 t2, Arrow t1' t2') →
                        unifyLoop (Map.insert t1 t1' (Map.insert t2 t2' (Map.fromList rest))) subst
                    (Product t1 t2, Product t1' t2') →
                        unifyLoop (Map.insert t1 t1' (Map.insert t2 t2' (Map.fromList rest))) subst
                    (Region s t, Region s' t') →
                        unifyLoop (Map.insert t t' (Map.fromList rest))
                                  (Map.insert s' (Stk s) subst)
                    (Reference s t, Reference s' t') →
                        unifyLoop (Map.insert t t' (Map.fromList rest))
                                  (Map.insert s' (Stk s) subst)
                    (Handle s, Handle s') →
                        unifyLoop (Map.fromList rest) (Map.insert s' (Stk s) subst)
                    (TyVar tau, _) →
                        if freeIn tau t'
                        then Fail
                        else let substT = Map.singleton tau (Type t')
                             in  unifyLoop (Map.fromList (apply substT rest))
                                           (Map.union substT subst)
                    (_, TyVar tau) →
                        if freeIn tau t'
                        then Fail
                        else let substT = Map.singleton tau (Type t')
                             in  unifyLoop (Map.fromList (apply substT rest))
                                           (Map.union substT subst)
                    otherwise → Fail
```

9 Further Work

Several points of potential further work have been identified. Firstly, our prototypal type inferencer is a definitional type-inferencer of all rules listed above, but no implementation of `f-RGN` is provided by this author. There are a few possible targets of compilation, including C and Rust, such that `f-RGN` could be compiled to in order to run efficiently.

Moreover, there are no lists, records, algebraic data types, or `let`-polymorphism in `f-RGN`. However, they could be easily added to an extension of `f-RGN`, with a more complex type inference algorithm.

Finally, the greatest virtue of regions are the possibility of parallelization. Concurrency and continuation-passing style computational primitives are also a possible extension to `f-RGN`.

10 Conclusion

Overall, `f-RGN` is a promising programming language for creating functional programs without memory leaks. By using a stack of regions and encapsulating all memory-transforming operations behind a monad, `f-RGN` effectively hides all changes in the memory state — which are the ultimate causes of memory unsafety and memory leaks. Moreover, `f-RGN` allows type inference using conventional algorithms and does not require explicit references to regions, making `f-RGN` friendly and approachable to the programmer.

11 Lessons Learned

Through studying type guarantees for memory, I find myself able to reason about type systems in a more intuitional fashion, and able to more effectively understand and evaluate arguments about the design of type systems. Moreover, since memory is ultimately a form of state, I gained a better understanding of states and the manipulation of it using monads and other type-theoretical tools (e.g., substructural typing). Finally, through designing and writing rules for `f-RGN`, I find myself more competent at creating sound typing rules and designing type systems.

Appendix

The rest of the type inference implementation in Haskell is listed below. The LaTeX source of this report is a valid Literate Haskell program; to extract Haskell code use the tool `unlit`.

Data type definitions:

```
type Id = String

data Kind = Stack | Star

data TyTerm = Type Type | Stk StackId deriving (Show)

type StackId = Id

data Type = TyVar Id
          | Integer
          | Bool
          | String
          | Arrow Type Type
          | Unit
          | Product Type Type
          | Region StackId Type
          | Reference StackId Type
          | Handle StackId
          deriving (Eq, Ord, Show)

data Expr = Variable Id
          | Literal Value
          | Lambda Id Expr
          | Apply Expr Expr
          | IfElse Expr Expr Expr
          | Empty
          | Pair Expr Expr
          | Unpack (Id, Id) Expr Expr
          | Return Expr
          | Then Expr Expr
          | New Expr Expr
          | Read Expr
          | Write Expr Expr
          | RunRgn Expr
          | LetRgn Expr Expr
          | NewStk
          | GetHnd Expr
          deriving (Eq, Ord, Show)

data Value = I Integer | S String | B Bool deriving (Eq, Ord, Show)

type Env a = Map.Map Id a
```

Utility functions for types:

```
eqType :: Type → Type → Bool
eqType (TyVar x) (TyVar y) = x == y
eqType Integer Integer = True
eqType Bool Bool = True
eqType String String = True
eqType Unit Unit = True
eqType (Arrow t1 t1') (Arrow t2 t2') = (eqType t1 t2) && (eqType t1' t2')
eqType (Product t1 t2) (Product t1' t2') = (eqType t1 t1') && (eqType t2 t2')
eqType (Region s1 t1) (Region s2 t2) = (s1 == s2) && (eqType t1 t2)
eqType (Reference s1 t1) (Reference s2 t2) = (s1 == s2) && (eqType t1 t2)
-- Note that this is actually not true, but here for the sake of simplicity.
-- As long as s1 < s2 or vice versa, hnd s1 and hnd s2 are equal types
eqType (Handle s1) (Handle s2) = (s1 == s2)
eqType _ _ = False

ftv :: Type → Set.Set Id
ftv (TyVar x) = Set.singleton x
ftv Integer = Set.empty
ftv Bool = Set.empty
ftv String = Set.empty
ftv Unit = Set.empty
ftv (Arrow t1 t2) = Set.union (ftv t1) (ftv t2)
ftv (Product t1 t2) = Set.union (ftv t1) (ftv t2)
ftv (Region s t) = Set.insert s (ftv t)
ftv (Reference s t) = Set.insert s (ftv t)
ftv (Handle s) = Set.singleton s

freeIn :: Id → Type → Bool
freeIn x tau = Set.member x (ftv tau)

class Substs a where
    apply :: TySubst → a → a

instance Substs Type where
    apply subst (TyVar x) = case Map.lookup x subst of
                                Nothing → TyVar x
                                Just (Type t)  → t
                                -- This shouldn't happen!
                                Just (Stk s) → TyVar x
    apply subst (Arrow t1 t2) = Arrow (apply subst t1) (apply subst t2)
    apply subst (Product t1 t2) = Product (apply subst t1) (apply subst t2)
    apply subst (Region s t) = case Map.lookup s subst of
                                  Nothing → Region s (apply subst t)
                                  Just (Stk s') → Region s' (apply subst t)
                                  -- Bad! Shouldn't happen
                                  Just _ → Region s (apply subst t)
    apply subst (Reference s t) = case Map.lookup s subst of
                                     Nothing → Reference s (apply subst t)
                                  Just (Stk s') → Reference s' (apply subst t)
                                  -- Bad! Shouldn't happen
                                  Just _ → Reference s (apply subst t)
  apply subst (Handle s) = case Map.lookup s subst of
                                Nothing → Handle s
                                Just (Stk s') → Handle s'
                                -- Bad! Shouldn't happen
                                Just _ → Handle s

instance Substs a ⇒ Substs [a] where
  apply subst = map (apply subst)

instance Substs a ⇒ Substs (a, a) where
  apply subst (a, b) = (apply subst a, apply subst b)

instance Substs TyConsSet where
  apply subst tcs = Map.fromList (map (apply subst) tcs')
    where tcs' = Map.toList tcs
```

Preserving monadic state in *inf*:

```
data InfEnv = InfEnv {}
data InfState = InfState {infSupply :: Int,
                          infStkSupply :: Int,
                          infSubst   :: TySubst}

type Inference a = ExceptT String (ReaderT InfEnv (StateT InfState IO)) a

runInf :: Inference a → IO (Either String a, InfState)
runInf t =
  do
    (result, st) ← runStateT (runReaderT (runExceptT t) initInfEnv) initInfState
    return (result, st)
  where
    initInfEnv = InfEnv{}
    initInfState = InfState{infSupply = 0, infStkSupply = 0, infSubst =
Map.empty}

freshTyVar :: String → Inference Type
freshTyVar prefix =
  do s ← get
     put s{infSupply = infSupply s + 1}
     return (TyVar (prefix ++ show (infSupply s)))

freshStkIdx :: String → Inference StackId
freshStkIdx prefix =
  do s ← get
     put s{infStkSupply = infStkSupply s + 1}
     return (prefix ++ show (infStkSupply s))
```

References

[1] Fluet, M., Morrisett, G., and Ahmed, A. Linear regions are all you need. In *Proceedings of the 15th European Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2006), ESOP'06, Springer-Verlag, pp. 7–21.

[2] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, J. W. Report on the fx-91 programming language. Tech. rep., DTIC Document, 1992.

[3] Tofte, M., and Talpin, J.-P. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1994), POPL '94, ACM, pp. 188–201.

[4] Turbak, F., Gifford, D., and Sheldon, M. *Design Concepts in Programming Languages.* MIT Press, 2008.