HASKELL SYNTAX AND STATIC SEMANTICS WRITTEN IN K-FRAMEWORK

Draft of November 24, 2018 at 21 : 20

BY

BRADLEY MORRELL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Elsa Gunter

ABSTRACT

For this thesis, I introduce a static semantics for Haskell by utilizing the K-Framework. This implementation includes support for the module system of Haskell but not for type classes. There are many layers that have to be implemented in K before type inference can be performed. The first part of the implementation is the entire context free syntax of Haskell in K. Since all the syntax is included, any program written in Haskell extended syntax can be parsed into an abstract syntax tree. However, this includes only the Haskell extended syntax but does not include the syntactic short-cuts such as treating tabs as syntactic sugar for grouping constructs such as curly braces. Programs that include multiple modules can be parsed, but the multiple modules must be written in a single file. This is unlike how the Glasgow Haskell Compiler allows for module imports, where each module must be kept in separate files. The multiple modules are then made as nodes in a directed acyclic graph. A directed edge in the graph represents a module importing another module. This graph is used for importing the user defined types from one module into another module. Context sensitive checks and type inference are then performed on modules. The static semantics specifies that, at each node in the graph, assuming all child modules are already checked and inferred, the user defined types of each of the child modules are imported into the module at the given node. All rules of the Haskell type system must take mutual recursion into account. There is repeated layering of inferences in Haskell. My semantics is mathematically precise and executable. I utilized the executability of the semantics to test both positive inferences and exceptional inferences. This is part of a larger project to give a formal semantics to Haskell.

Subject Keywords: Haskell; Type-System

To my parents, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CHAPTER 1

INTRODUCTION

A current problem with engineered systems is that the design of the system is not proven to be working. The system could be non-functional at design time. The designer may not fully understand the system environment or may have not considered the behavior of the system in rare circumstances. Then once the design is made, there may be bugs introduced by implementing the design incorrectly. The current way that programs are created is by making a design or a formal specification of the program, implementing it, and testing the program against unit tests or verifying the behavior of the program after the fact. Formal methods are ways to mathematically prove correctness of a system. Without formal methods, the only way to reason about a system is by testing it against different edge cases. Within the context of a programming language, one way a programming language can be formally specified is by defining a syntax and semantics for that language. The operational semantics of a programming language can be thought of as a transition system upon an abstract syntax tree, which is the program itself written in the language, and a state, which is a function from the variables in the tree to the current values of those variables. This way, real and complex programs written in natural looking programming languages can be interpreted as strings written in formal languages. Once a programming language is defined in this way, certain properties and behavior of the language and programs written in the language can be proven. K is a framework for creating the formal specification of a programming language. It then can interpret programs written in the language by running only the rules of the formal operational semantics of the programming language. This allows programs to be ran and analyzed formally. This way the formal specification of the complex programming language can be tested and analyzed with the use of a machine. A K-configuration defines the memory structure of the programming language, made up of cells. The program state can be thought of as the current values of the K-configuration at a certain point in time. Grammar can be written in K using the constructor 'syntax', and a semantic rule can be written in K using the constructor 'rule'. Haskell is a purely functional programming language with strong static typing. Purely functional means that the language only allows the user to make functions whose output is only dependant on the function input. Strong static typing means that before a program is run, a type inference algorithm checks the program and ensures that all functions and function applications are allowed with regards to the types of the inputs and outputs. Static refers to the fact that type inference is performed before the code is ran, and will not run during the runtime of the code. Strong typing refers to the fact that the compiler will not allow the user to perform workarounds like typecasting. This project details the syntax of Haskell and the type system of Haskell in K.

CHAPTER 2

CONTEXT FREE SYNTAX

The Haskell 2010 report is the current official specification of the Haskell language. The grammar specified in section 10.5 of the Haskell 2010 report is a specification of the expanded syntax of haskell. As specified in section 2.7, the expanded syntax of haskell specifies haskell programs when written using semicolons and braces. However, these can be omitted in a real haskell program. The compiler will then utilize layout rules for certain grammar structures instead. These are specified in section 10.3 The parser for this project does not implement these layout rules and instead only can parse the expanded, layout insensitive syntax of haskell. It would require another script to convert a program written using the layout sensitive syntax into the expanded syntax in order to parse the program. Haskell has a context free grammar. Section 10.1 specifies the notation used in the grammar. The notation of 10.1 are always in bold in the grammar. So

```
qvarid -> [ modid . ] varid
```

Means that

```
modid .
```

is optional, and the brackets

```
[ ]
```

are not part of the haskell code, but the period

```
.
```

is part of the haskell code. Any symbol that is not in bold needs to be written in the program in order to parse correctly. There are a lot of parts of the grammar that were tricky for me to implement in K. For instance, a sort definition with an optional part could be just written using a pipe

```
|
```

in the K syntax. So

```
qvarid -> [ modid . ] varid
```

Is written in my K syntax as

```
syntax QVarId ::= VarId | ModId "." VarId [klabel('qVarIdCon)]
```

However, an issue arises when you have something written like

```
data [context =>] simpletype [= constrs] [deriving]
```

As an option for the definition of the topdecl sort. What I did was, for each optional part, replace the optional part with a new sort. For instance, I replaced

```
[ context => ]
```

with a new sort called OptContext. Then I just specified

```
syntax OptContext ::= Context "=>" | "" [onlyLabel, klabel('emptyContext)]
```

```
// Syntax from haskell 2010 Report
// https://www.haskell.org/onlinereport/haskell2010/haskellch10.html#x17-17500010

module HASKELL-SYNTAX

    syntax Integer ::= Token{([0-9]+)
                | (([0][o]|[0][O])[0-7]+)
                | (([0][x] | [0][X])[0-9a-fA-F]+)}    [onlyLabel]

    syntax CusFloat ::= Token{([0-9]+[\.][0-9]+([e E][\+\-]?[0-9]+)?)
                              |([0-9]+[e E][\+\-]?[0-9]+)} [onlyLabel]
    syntax CusChar ::= Token{[\'](~[\'\\\&])[\']} [onlyLabel]
    syntax CusString ::= Token{[\"](~[\"\\]*)[\"]} [onlyLabel]

    syntax VarId ::= Token{[a-z\_][a-z A-Z\_0-9\']*} [onlyLabel] | "size" [onlyLabel]
```

I ran into another issue where a program with a variable called 'size' did not parse. I found out that this is because 'size' is a k keyword. So I just specified that a variable could be a variable token, or "size".

```
    syntax ConId ::= Token{[A-Z][a-zA-Z \_0-9\']*} [onlyLabel]
    syntax VarSym ::= Token{
   ([\! \# \$ \% \& \* \+ \/ \> \? \^][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]*)
  |[\-] | [\.]
  |([\.][\! \# \$ \% \& \* \+ \/ \< \= \> \? \@ \\ \^ \| \- \~ \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^
  | ([\-][\! \# \$ \% \& \* \+ \. \/ \< \= \? \@ \\ \^ \| \~ \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \
  | ([\@][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]+)
  | ([\~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]+)
  | ([\\][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]+)
  | ([\|][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]+)
  | ([\:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \
  | ([\<][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \~ \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \
  | ([\=][\! \# \$ \% \& \* \+ \. \/ \< \= \? \@ \\ \^ \| \~ \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \
    syntax ConSym ::= Token{[\:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \- \~][\! \# \$ \% \& \* \+ \.
[onlyLabel]

    syntax IntFloat ::= "(" Integer ")"  [bracket] //NOT OFFICIAL SYNTAX
                     | "(" CusFloat ")" [bracket]
```

I ran into an issue where floats and integers did not parse correctly. They caused parsing errors due to ambiguity of parsing. For example the number 123.45 had ambiguity where the parser did not know if 1, 12, or 123 where integers, and if 5 was an integer, or if the entire thing was one float. Normally in K, different tokens are separated with whitespaces. However, for some reason the parser had difficulty here. Initially, I added a workaround by requiring parentheses around each integer and floating point. f y z (2) This fixed the issue.

```
    syntax Literal ::= IntFloat | CusChar | CusString
    syntax TyCon ::= ConId
    syntax ModId ::= ConId | ConId "." ModId  [klabel('conModId)]
    syntax QTyCon ::= TyCon | ModId "." TyCon [klabel('conTyCon)]
    syntax QVarId ::= VarId | ModId "." VarId [klabel('qVarIdCon)]
    syntax QVarSym ::= VarSym | ModId "." VarSym  [klabel('qVarSymCon)]
    syntax QConSym ::= ConSym | ModId "." ConSym  [klabel('qConSymCon)]
/*    syntax QTyCls ::= QTyCon
    syntax TyCls ::= ConId
*/
    syntax TyVars ::= List{TyVar, ""} [klabel('typeVars)] //used in SimpleType syntax
    syntax TyVar ::= VarId
    syntax TyVarTuple ::= TyVar "," TyVar      [klabel('twoTypeVarTuple)]
                        | TyVar "," TyVarTuple [klabel('typeVarTupleCon)]

    syntax Con ::= ConId | "(" ConSym ")"   [klabel('conSymBracket)]
    syntax Var ::= VarId | "(" VarSym ")"   [klabel('varSymBracket)]
    syntax QVar ::= QVarId | "(" QConSym ")" [klabel('qVarBracket)]
    syntax QCon ::= QTyCon | "(" GConSym ")"  [klabel('gConBracket)]

    syntax QConOp ::= GConSym | "‘" QTyCon "‘"     [klabel('qTyConQuote)]
    syntax QVarOp ::= QVarSym | "‘" QVarId "‘"     [klabel('qVarIdQuote)]
    syntax VarOp ::= VarSym   | "‘" VarId "‘"      [klabel('varIdQuote)]
    syntax ConOp ::= ConSym   | "‘" ConId "‘"      [klabel('conIdQuote)]

    syntax GConSym ::= ":" | QConSym
    syntax Vars ::= Var
                  | Var "," Vars  [klabel('varCon)]
    syntax VarsType ::= Vars "::" Type  [klabel('varAssign)]
    syntax Ops ::= Op
                 | Op "," Ops  [klabel('opCon)]
    syntax Fixity ::= "infixl" | "infixr" | "infix"
    syntax Op ::= VarOp | ConOp
    syntax CQName ::= Var | Con | QVar

    /* syntax QConId ::= ConId | ModId "." ConId */

    syntax QOp ::= QVarOp | QConOp


    syntax ModuleName ::= "module" ModId [klabel('moduleName)]

    syntax Module ::= ModuleName "where" Body          [klabel('module)]
                    | ModuleName Exports "where" Body  [klabel('moduleExp)]
                    | Body                             [klabel('moduleBody)]

    syntax Body ::= "{" ImpDecls ";" TopDecls "}" [klabel('bodyimpandtop)]
                  | "{" ImpDecls "}" [klabel('bodyimpdecls)]
                  | "{" TopDecls "}" [klabel('bodytopdecls)]

    syntax ImpDecls ::= List{ImpDecl, ";"} [klabel('impDecls)]
```

The sort that contains all the other sorts is a module. A module represents one complete Haskell program. It can have either a name and a body, a name and a body with exports, or just a body.

```
module Foo where
{import Bar
}
```

This example program has a Module with only ImpDecls. It has one ImpDecl.

```
import Bar
```

This example program has a Module with no name and only ImpDecls. It has one ImpDecl.

```
syntax Exports ::= "(" ExportList OptComma ")"
syntax ExportList ::= List{Export, ","}

syntax Export ::= QVar
               | QTyCon OptCQList
               | ModuleName

//optional cname list
syntax OptCQList ::= "(..)"
                   | "(" CQList ")"    [klabel('cqListBracket)]
                     //Liyi: a check needs to place in preprocessing to check
                     //if the CQList is a cname list or a qvar list.
                   | "" [onlyLabel, klabel('emptyOptCNameList)]
syntax CQList ::= List{CQName, ","}

syntax ImpDecl ::= "import" OptQualified ModId OptAsModId OptImpSpec [klabel('impDecl)]
                 | "" [onlyLabel, klabel('emptyImpDecl)]
syntax OptQualified ::= "qualified"
                      | ""  [onlyLabel, klabel('emptyQualified)]
syntax OptAsModId ::= "as" ModId
                    | ""     [onlyLabel, klabel('emptyOptAsModId)]

syntax OptImpSpec ::= ImpSpec
                    | ""     [onlyLabel, klabel('emptyOptImpSpec)]

syntax ImpSpecKey ::= "(" ImportList OptComma ")"
syntax ImpSpec ::= ImpSpecKey
                 | "hiding" ImpSpecKey

syntax ImportList ::= List{Import, ","}

syntax Import ::= Var
               | TyCon CQList
```

An import is another module that this module depends on.

```
syntax TopDecls ::= List{TopDecl, ";"} [klabel('topdeclslist)]

syntax TopDecl ::= Decl [klabel('topdecldecl)]
                 > "type" SimpleType "=" Type [klabel('type)]
                 | "data" OptContext SimpleType OptConstrs OptDeriving  [klabel('data)]
                 | "newtype" OptContext SimpleType "=" NewConstr OptDeriving [klabel('newtype)]
                 | "class" OptContext ConId TyVar OptCDecls [klabel('class)]
                 | "instance" OptContext QTyCon Inst OptIDecls [klabel('instance)]
                 | "default" Types [klabel('default)]
                 | "foreign" FDecl [klabel('foreign)]
```

The main types of expressions in Haskell are TopDecls - Top Declarations. A top declaration can either be a type, a data, a newtype, a class, an instance, a default, a foreign, or an arbitrary declaration.

```
syntax FDecl ::= "import" CallConv CusString Var "::" FType
               | "import" CallConv Safety CusString Var "::" FType
               | "export" CallConv Safety CusString Var "::" FType
//Liyi: fdecl needs to use special function in preprocessing
// to get the actually elements from the impent and expent from the CusString
//did string analysis

syntax Safety ::= "unsafe" | "safe"

syntax CallConv ::= "ccall" | "stdcall" | "cplusplus" | "jvm" | "dotnet"
syntax FType ::= FrType
               | FaType "->" FType // unsure about this one syntax is ambiguous UNFINISHED

syntax FrType ::= FaType
                | "()"
```

```
syntax FaType ::= QTyCon ATypeList

//define declaration.
syntax OptDecls ::= "where" Decls | "" [onlyLabel, klabel('emptyOptDecls)]
syntax Decls ::= "{" DeclsList "}" [klabel('decls)]
syntax DeclsList ::= List{Decl, ";"} [klabel('declsList)]

syntax Decl ::= GenDecl
              | FunLhs Rhs [klabel('declFunLhsRhs)]
              | Pat Rhs [klabel('declPatRhs)]
```

Any sort that starts with 'Opt' means that this is optional. In K something can be made optional by declaring the necessary constructors or nothing.

A Decl is any general declaration. So something like

```
f x = x + 2
```

is a Decl.

```
syntax OptCDecls ::= "where" CDecls | "" [onlyLabel, klabel('emptyOptCDecls)]
syntax CDecls ::= "{" CDeclsList "}"
syntax CDeclsList ::= List{CDecl, ";"}

syntax CDecl ::= GenDecl
               | FunLhs Rhs
               | Var Rhs

syntax OptIDecls ::= "where" IDecls | "" [onlyLabel, klabel('emptyOptIDecls)]
syntax IDecls ::= "{" IDeclsList "}"
syntax IDeclsList ::= List{IDecl, ";"} [klabel('ideclslist)]

syntax IDecl ::= FunLhs Rhs [klabel('cdeclFunLhsRhs)]
               | Var Rhs [klabel('cdeclVarRhs)]
               | "" [onlyLabel, klabel('emptyIDecl)]

syntax GenDecl ::= VarsType
                 | Vars "::" Context "=>" Type   [klabel('genAssignContext)]
                 | Fixity Ops
                 | Fixity Integer Ops
                 | "" [onlyLabel, klabel('emptyGenDecl)]

//three optional data type for the TopDecl data operator.
//deriving data type
syntax OptDeriving ::= Deriving | "" [onlyLabel, klabel('emptyDeriving)]
syntax Deriving ::= "deriving" DClass
                  | "deriving" "(" DClassList ")"
syntax DClassList ::= List{DClass, ","}
syntax DClass ::= QTyCon

syntax FunLhs ::= Var APatList [klabel('varApatList)]
                | Pat VarOp Pat [klabel('patVarOpPat)]
                | "(" FunLhs ")" APatList [klabel('funlhsApatList)]

syntax Rhs ::= "=" Exp OptDecls [klabel('eqExpOptDecls)]
             | GdRhs OptDecls [klabel('gdRhsOptDecls)]

syntax GdRhs ::= Guards "=" Exp
               | Guards "=" Exp GdRhs
syntax Guards ::= "|" GuardList
syntax GuardList ::= Guard | Guard "," GuardList  [klabel('guardListCon)]
syntax Guard ::= Pat "<-" InfixExp
               | "let" Decls
               | InfixExp

//definition of exp
syntax Exp ::= InfixExp
             > InfixExp "::" Type  [klabel('expAssign)]
             | InfixExp "::" Context "=>" Type  [klabel('expAssignContext)]

syntax InfixExp ::= LExp
```

```
              > "-" InfixExp    [klabel('minusInfix)]
              > LExp QOp InfixExp

syntax LExp ::= AExp
              > "\\" APatList "->" Exp [klabel('lambdaFun)]
              | "let" Decls "in" Exp [klabel('letIn)]
              | "if" Exp OptSemicolon "then" Exp OptSemicolon "else" Exp [klabel('ifThenElse)]
              | "case" Exp "of" "{" Alts "}" [klabel('caseOf)]
              | "do" "{" Stmts "}" [klabel('doBlock)]
```

LExp is an important sort for the inference function. This is because LExp defines the different expression types which the inference function has specific rules for.

```
syntax OptSemicolon ::= ";" | "" [onlyLabel, klabel('emptySemicolon)]
syntax OptComma ::= "," | ""     [onlyLabel, klabel('emptyComma)]

syntax AExp ::= QVar [klabel('aexpQVar)]
              | GCon [klabel('aexpGCon)]
              | Literal [klabel('aexpLiteral)]
              > AExp AExp [left, klabel('funApp)]
              > QCon "{" FBindList "}"
              | AExp "{" FBindList "}" //aexp cannot be qcon UNFINISHED
                     //Liyi: first, does not understand the syntax, it is the Qcon {FBindlist}
                     //or QCon? Second, place a check in preprosssing.
                     //and also check the Fbindlist here must be at least one argument
              > "(" Exp ")"             [bracket]
              | "(" ExpTuple ")"
              | "[" ExpList "]"
              | "[" Exp OptExpComma ".." OptExp "]"
              | "[" Exp "|" Quals "]"
              | "(" InfixExp QOp ")"
              | "(" QOp InfixExp ")" //qop cannot be - (minus) UNFINISHED
                      //Liyi: place a check here to check if QOp is a minus
```

AExp is also an import sort for the inference function. The main parts of AExp that the inference function cares about is QVar and GCon.

```
syntax OptExpComma ::= "," Exp | "" [onlyLabel, klabel('emptyExpComma)]
syntax OptExp ::= Exp | "" [onlyLabel, klabel('emptyExp)]

syntax ExpList ::= Exp | Exp "," ExpList  [right]
syntax ExpTuple ::= Exp "," Exp           [right, klabel('twoExpTuple)]
                  | Exp "," ExpTuple      [right, klabel('expTupleCon)]

//constr datatypes
syntax OptConstrs  ::= "=" Constrs [klabel('nonemptyConstrs)] | "" [onlyLabel, klabel('emptyConstrs)]
syntax Constrs     ::= Constr [klabel('singleConstr)] | Constr "|" Constrs [klabel('multConstr)]
syntax Constr      ::= Con OptBangATypes [klabel('constrCon)] // (arity con = k, k    0) UNFINISHED
                     | SubConstr ConOp SubConstr
                     | Con "{" FieldDeclList "}"

syntax NewConstr   ::= Con AType [klabel('newConstrCon)]
                     | Con "{" Var "::" Type "}"

syntax SubConstr   ::= BType | "!" AType
syntax FieldDeclList ::= List{FieldDecl, ","}
syntax FieldDecl ::= VarsType
                   | Vars "::" "!" AType


syntax OptBangATypes ::= List{OptBangAType, " "} [klabel('optBangATypes)]
syntax OptBangAType ::= OptBang AType [klabel('optBangAType)]
syntax OptBang ::= "!" | "" [onlyLabel, klabel('emptyBang)]

syntax OptContext ::= Context "=>" | "" [onlyLabel, klabel('emptyContext)]
syntax Context ::= Class
```

```
                | "(" Classes ")"

syntax Classes ::= List{Class, ","}

syntax SimpleClass ::= QTyCon TyVar  [klabel('classCon)]

syntax Class       ::= SimpleClass
                     | QTyCon "(" TyVar ATypeList ")"
                          //Liyi: a check in preprossing to check if the Atype list is empty
                          //it must have at least one item

//define type and simple type
syntax SimpleType  ::= TyCon TyVars  [klabel('simpleTypeCon)]
syntax Type ::= BType
              | BType "->" Type  [klabel('typeArrow)]
syntax BType ::= AType
               | BType AType [klabel('baTypeCon)]

syntax ATypeList ::= List{AType, ""} [klabel('atypeList)]

syntax AType ::= GTyCon                        [klabel('atypeGTyCon)]
               | TyVar                         [klabel('atypeTyVar)]
               | "(" TypeTuple ")"             [klabel('atypeTuple)]
               | "[" Type "]"                  [klabel('tyList)]
               | "(" Type ")"                  [bracket]
syntax TypeTuple ::= Type "," Type             [right,klabel('twoTypeTuple)]
                   | Type "," TypeTuple        [klabel('typeTupleCon)]
syntax Types ::= List{Type, ","}

syntax GConCommas ::= "," | "," GConCommas
syntax GConCommon ::= "()" | "[]" | "(" GConCommas ")" //was incorrect syntax
syntax GTyCon ::= QTyCon
                | GConCommon
                | "(->)"

syntax GCon ::= GConCommon
              | QCon

//inst definition
syntax Inst  ::= GTyCon
               | "(" GTyCon TyVars ")" //TyVars must be distinct UNFINISHED
               | "(" TyVarTuple ")" //TyVars must be distinct
               | "[" TyVar "]"  [klabel('tyVarList)]
               | "(" TyVar "->" TyVar ")" //TyVars must be distinct
//pat definition
syntax Pat ::= LPat QConOp Pat
             | LPat

syntax LPat ::= APat
              | "-" IntFloat    [klabel('minusPat)]
              | GCon APatList  [klabel('lpatCon)]// arity gcon = k UNFINISHED

syntax APatList ::= APat | APat APatList [klabel('apatCon)]

syntax APat ::= Var [klabel('apatVar)]
              | Var "@" APat
              | GCon
              | QCon "{" FPats "}"
              | Literal [klabel('apatLiteral)]
              | "_"
              | "(" Pat ")"   [bracket]
              | "(" PatTuple ")"
              | "[" PatList "]"
              | "~" APat

syntax PatTuple ::= Pat "," Pat           [klabel('twoPatTuple)]
                  | Pat "," PatTuple      [klabel('patTupleCon)]
syntax PatList ::= Pat
                 | Pat "," PatList        [klabel('patListCon)]

syntax FPats ::= List{FPat, ","}
syntax FPat ::= QVar "=" Pat
```

```
//definition of quals
syntax Quals ::= Qual | Qual "," Quals  [klabel('qualCon)]

syntax Qual ::= Pat "<-" Exp
              | "let" Decls
              | Exp

//definition of alts
syntax Alts ::= Alt | Alt ";" Alts

syntax Alt ::= Pat "->" Exp  [klabel('altArrow)]
             | Pat "->" Exp "where" Decls
             | "" [onlyLabel, klabel('emptyAlt)]

//definition of stmts
syntax Stmts ::= StmtList Exp OptSemicolon
syntax StmtList ::= List{Stmt, ""}
syntax Stmt ::= Exp ";"
              | Pat "<-" Exp ";"
              | "let" Decls ";"
              | ";"

 //definition of fbind
syntax FBindList ::= List{FBind, ","}
syntax FBind ::= QVar "=" Exp
```

CHAPTER 3

CONFIGURATION

```
requires "haskell-syntax.k"

module HASKELL-CONFIGURATION
    imports HASKELL-SYNTAX

    syntax KItem ::= "startImportRecursion"
    syntax KItem ::= callInit(K)
    //syntax KItem ::= initPreModule(K) [function]
    //syntax KItem ::= tChecker(K) [function]

    configuration
        <T>
            <k> $PGM:ModuleList ~> startImportRecursion </k>
            <tempModule> .K </tempModule>
            <tempCode> .K </tempCode>
```

The k cell is the cell that computation takes place in. tempModule is the name of the current module.

```
            <typeIterator> 1 </typeIterator>
```

typeIterator is used for creating a fresh type variable. It has the current count of how many fresh type variables that were created.

```
            <tempAlpha> .K </tempAlpha>
            <tempAlphaMap> .Map </tempAlphaMap>
```

Alpha is a map of type renamings. So if a user declares

```
data MyBool = TTrue
;type MyBooltwo = MyBool
```

Then MyBooltwo is a renaming of MyBool. In tempAlpha, an AObject is made. An AObject is a KItem with two children. One can be thought of as a Key and the other is the Value for a map. So MyBool -¿ MyBooltwo. However, we want to check and reject programs that have multiple renamings, so we cannot use a K Map which has idempotence. However, once we make this check, we can then use a K Map. This is what tempAlphaMap is.

```
            <tempBeta> .Map </tempBeta>
            <tempT> .K </tempT>
            <tempDelta> .Map </tempDelta>
            <tempAlphaStar> .K </tempAlphaStar>
            <tempBetaStar> .K </tempBetaStar>
            <importTree> .List </importTree>
            <recurImportTree> .List </recurImportTree>
            <impTreeVMap> .Map </impTreeVMap>
            <modules> //static information about a module
                <module multiplicity="*">
```

```
                    <moduleName> .K </moduleName>
                    <moduleAlphaStar> .K </moduleAlphaStar>
                    <moduleBetaStar> .K </moduleBetaStar>
                    <moduleImpAlphas> .List </moduleImpAlphas>
                    <moduleImpBetas> .List </moduleImpBetas>
                    <moduleCompCode> .K </moduleCompCode>
                    <moduleTempCode> .K </moduleTempCode>
                    <imports> .Set </imports>
                    <classes> //static information about a module
                        <class multiplicity="*">
                            <className> .K </className>
                        </class>
                    </classes>
                </module>
            </modules>
        </T>

endmodule
```

CHAPTER 4

CONTEXT SENSITIVE CHECKS

I also placed the user defined types into data structures in order to perform several checks to make sure that the user did not have errors when creating types. Then the data structures will be transformed into a form that will be used for type inferencing. Section 4 of the Haskell 2010 report specifies the haskell type system. In the topdecl sort, there are three typecons that are used to create user defined datatypes. Data, type, and newtype. The end goal is to put the user defined types into a data structure which I can use to perform type inferencing. These three typecons are used to create user defined types. The first one is type,

```
type simpletype = type
```

This is used in a haskell program to declare a new type as a single type. In effect, it renames the type where both names now can be used to refer to the type.

```
type Username = String
```

Is one such example usage of type, it creates a new type Username, which is defined as just a string. Now the programmer can refer to Username or String to make a string. The second one is data,

```
data [context =>] simpletype [= constrs] [deriving]
```

This allows a user to declare a new type that may include many fields, and polymorphic types. For instance:

```
data Date = Date Int Int Int
```

This is a new type that includes the typecon 'Date' followed by three integers.

```
data Poly a = Number a
```

This is a new polymorphic type with polymorphic parameter a, that has the typecon 'Number'. The third one is newtype,

```
newtype [context =>] simpletype = newconstr [deriving]
```

This is very similar to data except it only parses when the newtype has only one typecon and one field.

I perform several checks here, 1. The programmer should not be able to make two user defined datatypes with the same name, even if one is created using 'data' and another is created using 'type' for instance. 2. The programmer should not be able to use the same typecons when making different options for their types or use the same typecons for different types. 3. There should be no cycles in type renaming using 'type', and the type renaming chains using 'type' should terminate with a type defined with 'data' or 'newtype'. 4. The argument sorts for types defined using 'data' or 'newtype' should be types that exist. 5. The polymorphic parameters that appear on the right hand side of a 'data' declaration need to appear on the left hand side as well. 6. The polymorphic parameters that appear on the left hand side of a 'data' declaration need to be unique. I implemented a map, called alpha, of new type names as the keys and their declared types as the entries. I then collected all appearances of the typecon 'type' in the program, and put simpletype -¿ type in the alpha map. However, one of the things I needed to check for in the program was whether a user declared multiple definitions with 'type', so I could not use a map in K because they only allow unique keys with unique entries. So I initially used a set of tuples, and then changed it to a map after checking for multiple type declarations.

The second data structure I made is called T. T holds the user defined types created using 'data' and 'newtype'.

```
syntax KItem ::= TList(K)
//list of T objects for every new type introduced by data and newtype
syntax KItem ::= TObject(K,K,K)
//(type name, entire list of poly type vars, list of inner T pieces)
syntax KItem ::= InnerTPiece(K,K,K,K,K)
//(type constructor, poly type vars, argument sorts, entire constr block, type name)
```

T is a list of TObjects, each TObject represents a single user defined datatype. It holds the name, the list of polymorphic parameters, and a list of inner T pieces. An inner T piece represents an option of what a type could be. It consists of a type constructor, a list of polymorphic parameters required for this option, the fields for this option, the entire subtree of the AST for this option unedited, and the type name again. I then used these data structures to perform these checks, and afterwards will transform them into a new data structure to perform type inferencing.

```
//
requires "haskell-syntax.k"
requires "haskell-configuration.k"

module HASKELL-PREPROCESSING
    imports HASKELL-SYNTAX
    imports HASKELL-CONFIGURATION
```

```
    //USER DEFINED LIST
    //definition of ElemList

    //syntax KItem ::= ElemList
    syntax ElemList ::= List{Element,","} [strict]
//    syntax Int ::= lengthOfList(ElemList) [function]

//    rule lengthOfList(.ElemList) => 0
//    rule lengthOfList(val(K:K),L:ElemList) => 1 +Int lengthOfList(L)
//    rule lengthOfList(valValue(K:K),L:ElemList) => 1 +Int lengthOfList(L)

    syntax Element ::= val(K) [strict]
    syntax ElementResult ::= valValue(K)
    syntax Element ::= ElementResult
    syntax KResult ::= ElementResult
    rule val(K:KResult) => valValue(K) [structural]

    //form ElemList
//    syntax ElemList ::= formElemList(K) [function]

    //CONVERT ~> TO List
    //list convert
//    syntax List ::= convertToList(K)  [function]
//    rule convertToList(.K) => .List
//    rule convertToList(A:KItem ~> B:K) => ListItem(A) convertToList(B)


    syntax KItem ::= dealWithImports(K,K)

    rule <k> 'modListSingle('module(A:K,, B:K)) => dealWithImports(A,B) ...</k>

    (.Bag =>
         <module>...    //DOT DOT DOT MEANS OVERWRITE ONLY SOME OF THE DEFAULTS
           <moduleName> A </moduleName>
      ...</module>
    )

    rule <k> 'modList('module(A:K,, B:K),, C:K) => dealWithImports(A,B) ~> C ...</k>

    (.Bag =>
         <module>...    //DOT DOT DOT MEANS OVERWRITE ONLY SOME OF THE DEFAULTS
           <moduleName> A </moduleName>
      ...</module>
    )

//    rule dealWithImports(Mod:K, A:K) => callInit(A)

//    rule <k> dealWithImports(Mod:K, A:K) => callInit(A) ...</k>

    rule <k> dealWithImports(Mod:K, 'bodyimpandtop(A:K,, B:K)) => .K ...</k>
        <importTree> L:List => L importListConvert(Mod, A) </importTree>
        <recurImportTree> L:List => L importListConvert(Mod, A) </recurImportTree>

        <moduleName> Mod </moduleName>
        <imports> S:Set (.Set => SetItem(A)) </imports>
        <moduleTempCode> OldTemp:K => B </moduleTempCode>

    rule <k> dealWithImports(Mod:K, 'bodyimpdecls(A:K)) => .K ...</k>
        <importTree> L:List => L importListConvert(Mod, A) </importTree>
        <recurImportTree> L:List => L importListConvert(Mod, A) </recurImportTree>

        <moduleName> Mod </moduleName>
        <imports> S:Set (.Set => SetItem(A)) </imports>

//    rule <k> dealWithImports(Mod:K, 'bodytopdecls(A:K)) => callInit(A) ...</k>
    rule <k> dealWithImports(Mod:K, 'bodytopdecls(B:K)) => .K ...</k>

        <moduleName> Mod </moduleName>
        <moduleTempCode> OldTemp:K => B </moduleTempCode>

    //importlist convert
    syntax List ::= importListConvert(K,K) [function]
```

```
    syntax KItem ::= impObject(K,K)

    rule importListConvert(Name:K, 'impDecls(A:K,, Rest:K)) => importListConvert(Name, A) importListConvert(Name, 
    rule importListConvert('moduleName(Name:K), 'impDecl(A:K,, Modid:K,, C:K,, D:K)) => ListItem(impObject(Name, M
    rule importListConvert(Name:K, .ImpDecls) => .List

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    /*NEW TODO ALGORITHM
1. Construct tree for module inclusion
2. Check tree for cycles
3. Go to each leaf and recursively go up the tree and build alpha* and beta* for the types of the module and the c
(and specify scoping) (desugar the scope so that each type specifies the scope) */

    syntax KItem ::= "checkImportCycle"
    syntax KItem ::= "recurseImportTree"

/*     rule <k> performNextChecks
            => checkUseVars
               ~> (checkLabelUses
               ~> (checkBlockAddress(.K)
               ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS -Set TES)
               ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
               ~> (checkAllExpInFromInvoke(.K, TES)
               ~> (checkLandingpad
               ~> checkLandingDomResumes)))))) ...</k> */

    rule <k> startImportRecursion => checkImportCycle
                                     ~> (recurseImportTree)...</k>

    syntax KItem ::= cycleCheck(K,Map,List,List) [function] //current node, map of all nodes to visited or not, sta
    syntax Map ::= createVisitMap(List,Map) [function] //graph, visitmap
    syntax KItem ::= getUnvisitedNode(K,K, Map) [function] //visitmap
    syntax List ::= getNodeNeighbors(K,List) [function] //visitmap

    rule <k> checkImportCycle
            => cycleCheck(.K,createVisitMap(I, .Map),.List,I) ...</k>
         <importTree> I:List </importTree>
         <impTreeVMap> .Map => createVisitMap(I, .Map) </impTreeVMap>

    syntax KItem ::= "visited"
    syntax KItem ::= "unvisited"
    syntax KItem ::= "none"

    rule createVisitMap(ListItem(impObject(A:K,B:K)) Rest:List, M:Map)
            => createVisitMap(Rest, M[A <- unvisited][B <- unvisited])
    rule createVisitMap(.List, M:Map) => M

    rule getUnvisitedNode(.K, .K, .Map) => none
    rule getUnvisitedNode(.K, .K, (A:K |-> B:K) M:Map)
          => getUnvisitedNode(A, B, M)
    rule getUnvisitedNode(A:KItem, unvisited, M:Map) => A
    rule getUnvisitedNode(A:KItem, visited, M:Map)
           => getUnvisitedNode(.K, .K, M)



    rule getNodeNeighbors(Node:K,.List) => .List
    rule getNodeNeighbors(.K,Rest:List) => .List

    rule getNodeNeighbors(Node:KItem,ListItem(impObject(Node,B:KItem)) Rest:List) => getNodeNeighbors(Node, Rest) 
    rule getNodeNeighbors(Node:KItem,ListItem(impObject(A:KItem,B:KItem)) Rest:List) => getNodeNeighbors(Node, Res
         requires Node =/=K A


    rule cycleCheck(none, M:Map, .List, L:List) => .K
    rule cycleCheck(.K, M:Map, .List, I:List) => cycleCheck(getUnvisitedNode(.K, .K, M), M, .List, I)
    rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:List) => cycleCheck(Node, M, S, I)
    rule cycleCheck(Node:K, M:Map, S:List, I:List)
              => cycleCheck(.K, M[Node <- visited], getNodeNeighbors(Node,I) S, I)
         requires Node =/=K .K andBool Node =/=K none
    rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) => cycleCheck(Node, M, S, I)
         requires S =/=K .List
```

```
/*
    rule cycleCheck(A:K,.K,.K,I:K) => cycleCheck(A,createVisitMap(I,.Map),.List,I)



    rule cycleCheck(Node:K, M:Map, S:List, I:K) => cycleCheck(.K, M[Node <- visited], getNodeNeighbors(Node,I) S, I

    rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) => cycleCheck(Node, M, S, I)
    //rule cycleCheck(.K, M:Map, .K, ListItem(impObject(A:K,B:K)) Rest:List) => cycleCheck(ListItem(impObject(A:K,I
*/

////////////////////////////////////////////////////////////////////////////////////////////////////////

//COPY IMPORT GRAPH, NEED SECOND GRAPH FOR RECURSING, ADDITIONAL GRAPH FOR SELECTING IMPORTS FOR ALPHA* AND BETA*
//DFS for leaf
//acquire alpha and beta for leaf
//merge alpha and beta with imports to produce alpha* and beta*
//perform checks
//perform inferencing
//insert alpha* and beta* into importing modules
//remove all edges pointing to leaf

    syntax KItem ::= "leafDFS"
    syntax KItem ::= "getAlphaAndBeta"
    syntax KItem ::= "getAlphaBetaStar"
    syntax KItem ::= "performIndividualChecks"
    syntax KItem ::= "performIndividualInferencing"
    syntax KItem ::= "insertAlphaBetaStar"
    syntax KItem ::= "removeAllEdges"
    syntax KItem ::= "seeIfFinished"

    rule <k> recurseImportTree => leafDFS
                                  ~> (getAlphaAndBeta
                                  //~> (getAlphaBetaStar
                                  ~> (performIndividualInferencing))...</k>

//rule <k> dealWithImports(Mod:K, 'bodytopdecls(A:K)) => callInit(A) ...</k>

//    rule <k> leaf
//             => cycleCheck(.K,createVisitMap(I, .Map),.List,I) ...</k>
//         <importTree> I:List </importTree>

////////////////////////////////////////////////////////////////////////////////////////////////////////

    syntax KItem ::= returnLeafDFS(K,List,Map) [function] //current node, map of all nodes to visited or not, stack
    syntax KItem ::= innerLeafDFS(K,List) [function]
    syntax KItem ::= loadModule(K)

    rule <k> leafDFS
             => returnLeafDFS(.K,I,M) ...</k>
         <recurImportTree> I:List </recurImportTree>
         <impTreeVMap> M:Map </impTreeVMap>

    rule returnLeafDFS(.K,ListItem(impObject(Node:KItem,B:KItem)) I:List,M:Map) => returnLeafDFS(B,I,M)
    rule returnLeafDFS(Node:KItem,I:List,M:Map) => returnLeafDFS(innerLeafDFS(Node,I),I,M)
         requires innerLeafDFS(Node,I) =/=K none
    rule returnLeafDFS(Node:KItem,I:List,M:Map) => loadModule(Node)
         requires innerLeafDFS(Node,I) ==K none

    rule innerLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem)) I:List) => B
    rule innerLeafDFS(Node:KItem,ListItem(impObject(A:KItem,B:KItem)) I:List) => innerLeafDFS(Node,I)
         requires Node =/=K A
    rule innerLeafDFS(Node:KItem,.List) => none
//    returnLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem)) I:List,M:Map) => returnLeafDFS(B,I,M)


////////////////////////////////////////////////////////////////////////////////////////////////////////

    //call before Checker Code
//    rule <k> callInit(S:K) => initPreModule(S) ...</k>
//         <tempModule> A:K => S </tempModule>
```

```
    rule <k> loadModule(S:KItem) => .K ...</k>
         <tempModule> A:K => S </tempModule>

    rule <k> getAlphaAndBeta => initPreModule(Code) ...</k>
         <tempModule> Mod:KItem </tempModule>

         <moduleName> 'moduleName(Mod) </moduleName>
         <moduleTempCode> Code:KItem </moduleTempCode>


////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //get alpha and beta
    syntax KItem ::= Module(K, K)
    syntax KItem ::= preModule(K,K) //(alpha, T)

    // STEP 1 CONSTRUCT T AND ALPHA
    // alpha = type
    // T = newtype and data, temporary data structure

    syntax KItem ::= initPreModule(K) [function]
    syntax KItem ::= getPreModule(K, K) [function] //(Current term, premodule)
    syntax KItem ::= makeT (K,K,K,K)

    syntax KItem ::= fetchTypes (K,K,K,K)

    syntax List ::= makeInnerT (K,K,K) [function] //LIST
    syntax List ::= getTypeVars(K) [function] //LIST

    syntax KItem ::= getCon(K) [function]
    syntax List ::= getArgSorts(K) [function] //LIST

    syntax KItem ::= AList(K)
    syntax KItem ::= AObject(K,K) //(1st -> 2nd) map without idempotency
    syntax KItem ::= ModPlusType(K,K)

    syntax KItem ::= TList(K) //list of T objects for every new type introduced by data and newtype
    syntax KItem ::= TObject(K,K,K,K) //(module name, type name, entire list of poly type vars, list of inner T pi
    syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type constructor, poly type vars, argument sorts, entire constr bloc

//    rule initPreModule('module(I:ModuleName,, J:K)) => getPreModule(J,preModule(AList(.List),TList(.List)))
//    rule initPreModule('moduleExp(I:ModuleName,, L:K,, J:K)) => getPreModule(J,preModule(AList(.List),TList(.List
//    rule initPreModule('moduleBody(J:Body)) => getPreModule(J,preModule(AList(.List),TList(.List)))

    rule initPreModule(J:K) => getPreModule(J,preModule(AList(.List),TList(.List)))

    rule getPreModule('bodytopdecls(I:K), J:K) => getPreModule(I,J)
    rule getPreModule('topdeclslist('type(A:K,, B:K),, Rest:K),J:K) => fetchTypes(A,B,Rest,J) //constructalpha


    rule getPreModule('topdeclslist('data(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => makeT(B,C,Rest,J)
    rule getPreModule('topdeclslist('newtype(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => makeT(B,C,Rest,J)


    rule getPreModule('topdeclslist('topdecldecl(A:K),, Rest:K),J:K) => getPreModule(Rest,J)
    rule getPreModule('topdeclslist('class(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => getPreModule(Rest,J)
    rule getPreModule('topdeclslist('instance(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => getPreModule(Rest,J)
    rule getPreModule('topdeclslist('default(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => getPreModule(Rest,J)
    rule getPreModule('topdeclslist('foreign(A:K,, B:K,, C:K,, D:K),, Rest:K),J:K) => getPreModule(Rest,J)
    rule getPreModule(.TopDecls,J:K) => J

    //rule getPreModule('module(I:ModuleName,L:K, J:K)) => preModule(J)

    rule <k> fetchTypes('simpleTypeCon(I:TyCon,, H:TyVars), 'atypeGTyCon(C:K), Rest:K, preModule(AList(M:List), L:K
         <tempModule> ModName:KItem </tempModule>

    rule <k> makeT('simpleTypeCon(I:TyCon,, H:TyVars), D:K, Rest:K, preModule(AList(M:List), TList(ListInside:List
         <tempModule> ModName:KItem </tempModule>

    rule makeInnerT(A:K,B:K,'nonemptyConstrs(C:K)) => makeInnerT(A,B,C)
    rule makeInnerT(A:K,B:K,'singleConstr(C:K)) => ListItem(InnerTPiece(getCon(C),getTypeVars(C),getArgSorts(C),C,A
    rule makeInnerT(A:K,B:K,'multConstr(C:K,, D:K)) => ListItem(InnerTPiece(getCon(C),getTypeVars(C),getArgSorts(C

    rule getTypeVars('constrCon(A:K,, B:K)) => getTypeVars(B)
```

```
    rule getTypeVars('optBangATypes(A:K,, Rest:K)) => getTypeVars(A) getTypeVars(Rest)
    rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K)) => getTypeVars(Rest)
    rule getTypeVars('atypeGTyCon(A:K)) => .List
    rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
    rule getTypeVars(.OptBangATypes) => .List

    //rule getCon('emptyConstrs()) => .K
    //rule getCon('nonemptyConstrs(A:K)) => getCon(A)
    rule getCon('constrCon(A:K,, B:K)) => A

    //rule getArgSorts('constrCon(A:K,, B:K)) => B
    rule getArgSorts('constrCon(A:K,, B:K)) => getArgSorts(B)
    rule getArgSorts('optBangATypes(A:K,, Rest:K)) => getArgSorts(A) getArgSorts(Rest)
    rule getArgSorts('optBangAType('emptyBang(.KList),, Rest:K)) => getArgSorts(Rest)
    rule getArgSorts('atypeGTyCon(A:K)) => ListItem(A)
    rule getArgSorts('atypeTyVar(A:K)) => .List
    rule getArgSorts(.OptBangATypes) => .List

///////////////////////////////////////////////////////////////////////////////////////////////////////////

    rule <k> preModule(A:K,T:K) => startTTransform ...</k>
         <tempAlpha> OldAlpha:K => A </tempAlpha>
         <tempT> OldT:K => T </tempT>

///////////////////////////////////////////////////////////////////////////////////////////////////////////

    // STEP 2 PERFORM CHECKS

    syntax KItem ::= "error"

    syntax KItem ::= "startChecks"
    syntax KItem ::= "checkNoSameKey"
        //Keys of alpha and keys of T should be unique
    syntax KItem ::= "checkTypeConsDontCollide"
        //Make sure typeconstructors do not collide in T
    syntax KItem ::= "makeAlphaMap"
        //make map for alpha
    syntax KItem ::= "checkAlphaNoLoops"
        //alpha check for no loops
        //check alpha to make sure that everything points to a T
    syntax KItem ::= "checkArgSortsAreTargets"
           //Make sure argument sorts [U] [W,V] are in the set of keys of alpha and targets of T, (keys of T)
    syntax KItem ::= "checkParUsed"
//NEED TO CHECK all the polymorphic parameters from right appear on left. RIGHT SIDE ONLY
//NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY

//    rule <k> preModule(A:K,T:K) => startChecks ...</k>
//         <tempAlpha> OldAlpha:K => A </tempAlpha>
//         <tempT> OldT:K => T </tempT>


/*    rule <k> performNextChecks
            => checkUseVars
               ~> (checkLabelUses
               ~> (checkBlockAddress(.K)
               ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS -Set TES)
               ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
               ~> (checkAllExpInFromInvoke(.K, TES)
               ~> (checkLandingpad
               ~> checkLandingDomResumes)))))) ...</k> */

    rule <k> startChecks
            => checkNoSameKey
               ~> (checkTypeConsDontCollide
               ~> (makeAlphaMap
               ~> (checkAlphaNoLoops
               ~> (checkArgSortsAreTargets
               ~> (checkParUsed))))) ...</k>

    rule <k> checkTypeConsDontCollide
            => tyConCollCheck(T,.List,.Set) ...</k>
         <tempT> T:K </tempT>
```

```
    //syntax KItem ::= tChecker(K) [function]
    syntax KItem ::= tyConCollCheck(K,K,K) [function] //(TList,List of Tycons,Set of Tycons)
    syntax KItem ::= lengthCheck(K,K) [function]
    //syntax KItem ::= tyConCollCheck(K,K,K) [function]
    //syntax K ::= innerCollCheck(K) [function]
    //syntax K ::= tyConCollCheckPasser(K, K) [function]

    //rule tChecker(preModule(Alpha:Map,T:K,Mod:K)) => tyConCollCheck(innerCollCheck(T),preModule(Alpha,T,Mod))

    //rule tyConCollCheck(.K,preModule(Alpha:Map,H:K,Mod:K)) => tyConCollCheck(innerCollCheck(H),preModule(Alpha,H

    rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:K,ListItem(InnerTPiece(Ty:K,E:K,F:K,H:K,G:K)) Inne
                tyConCollCheck(TList(ListItem(TObject(ModName,A,B,Inners)) Rest),ListItem(Ty) J, SetItem(Ty) D
    rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:K,.List)) Rest:List),J:List,D:Set) =>
                tyConCollCheck(TList(Rest),J,D)
    rule tyConCollCheck(TList(.List),J:List,D:Set) =>
                lengthCheck(size(J),size(D))

    rule lengthCheck(A:Int, B:Int) => .K
                requires A ==Int B

    rule lengthCheck(A:Int, B:Int) => error
                requires A =/=Int B

    //rule tyConCollCheck(TList(TObject(A:K,B:K,C:K) ~> Rest:K),J:K) => tyConCollCheckPasser(TList(innerCollCheck(
    syntax KItem ::= keyCheck(K,K,K,K) [function] //(Alpha, T, List of names, Set of names)

    rule <k> checkNoSameKey
           => keyCheck(A, T, .Set, .List) ...</k>
         <tempAlpha> A:K </tempAlpha>
         <tempT> T:K </tempT>
    //rule <k> checkAlphaNoSameKey
    //         => akeyCheck(.K, .Set) ...</k>

    rule keyCheck(AList(ListItem(AObject(A:K,B:K)) C:List), T:K, D:Set, G:List) => keyCheck(AList(C), T, SetItem(A
    rule keyCheck(AList(.List), TList(ListItem(TObject(ModName:K, A:K,B:K,C:K)) Rest:List), D:Set, G:List) => keyC
    rule keyCheck(AList(.List), TList(.List), D:Set, G:List) => lengthCheck(size(G),size(D))


    syntax KItem ::= makeAlphaM(K,K) [function] //(Alpha, AlphaMap)
    syntax KItem ::= tAlphaMap(K) //(AlphaMap) temp alphamap

    rule <k> makeAlphaMap
           => makeAlphaM(A, .Map) ...</k>
         <tempAlpha> A:K </tempAlpha>

    rule makeAlphaM(AList(ListItem(AObject(A:K,B:K)) C:List), M:Map) => makeAlphaM(AList(C), M[A <- B])
    rule makeAlphaM(AList(.List), M:Map) => tAlphaMap(M)

    rule <k> tAlphaMap(M:K) => .K ...</k>
         <tempAlphaMap> OldAlphaMap:K => M </tempAlphaMap>

//    syntax KItem ::= tkeyCheck(K,K,K) [function] //(T,List of T,Set of T)

//    rule <k> checkTNoSameKey
//           => tkeyCheck(T, .Set, T) ...</k>
//        <tempT> T:K </tempT>

//    rule tkeyCheck(TList(ListItem(TObject(A:K,B:K,C:K)) Rest:List), D:Set, G:K) => tkeyCheck(TList(Rest), SetIten
//    rule tkeyCheck(TList(.List), D:Set, TList(G:List)) => lengthCheck(size(G),size(D))

    syntax KItem ::= aloopCheck(K,K,K,K,K,K,K) [function] //(Alpha,List of Alpha,Set of Alpha,CurrNode,lengthcheck

    rule <k> checkAlphaNoLoops
           => aloopCheck(A,.List,.Set,.K,.K,T,.Set) ...</k>
         <tempAlphaMap> A:K </tempAlphaMap>
         <tempT> T:K </tempT>

    //aloopCheck set and list to check cycles
    rule aloopCheck(Alpha:Map (A:KItem |-> B:KItem), D:List, G:Set, .K, .K,T:K,S:Set) => aloopCheck(Alpha, ListIten
    rule aloopCheck(Alpha:Map (H |-> B:KItem), D:List, G:Set, H:KItem, .K,T:K,S:Set) => aloopCheck(Alpha, ListItem

    rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:K,S:Set) => aloopCheck(Alpha, .List, .Set, .K, lengthC
```

```
        requires (notBool H in keys(Alpha)) andBool (H in typeSet(T, .Set) orBool H in S)

    rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:K,S:Set) => error //terminal alpha rename is not in T K
        requires (notBool H in keys(Alpha)) andBool (notBool (H in typeSet(T, .Set) orBool H in S))


    syntax Set ::= typeSet(K,K) [function] //(K, KSet)
    rule typeSet(TList(ListItem(TObject(ModName:K, A:K,B:K,C:K)) Rest:List), D:Set) => typeSet(TList(Rest), SetItem
    rule typeSet(TList(.List), D:Set) => D

//    rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K) => keys(Alpha) ~> H
//        requires notBool H in keys(Alpha)

    rule aloopCheck(.Map, .List, .Set, .K, .K,T:K, S:Set) => .K
//    rule aloopCheck(AList(Front:List ListItem(AObject(H,B:K)) C:List), D:List, G:Set, H:ConId) => aloopCheck(ALi


//    syntax KItem ::= TList(K) //list of T objects for every new type introduced by data and newtype
//    syntax KItem ::= TObject(K,K,K) //(type name, entire list of poly type vars, list of inner T pieces)
//    syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type constructor, poly type vars, argument sorts, entire constr b

//Make sure argument sorts [U] [W,V] are in the set of keys of alpha and targets of T, (keys of T)

    syntax KItem ::= argSortCheck(K,K,K) [function] //(T,AlphaMap)

    rule <k> checkArgSortsAreTargets
            => argSortCheck(T,A,typeSet(T,.Set)) ...</k>
        <tempAlphaMap> A:K </tempAlphaMap>
        <tempT> T:K </tempT>

    rule argSortCheck(TList(ListItem(TObject(ModName:K, A:K,B:K,ListItem(InnerTPiece(C:K,D:K,ListItem(Arg:KItem) A
        requires ((Arg in keys(AlphaMap)) orBool (Arg in Tset))

    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,D:K,ListItem(Arg:KItem) Ar
        requires (notBool ((Arg in keys(AlphaMap)) orBool (Arg in Tset)))

    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,D:K,.List,E:K,F:K)) InnerR

    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,.List)) TListRest:List),AlphaMap:Map,Tset:Set) => a

    rule argSortCheck(TList(.List),AlphaMap:Map,Tset:Set) => .K

//NEED TO CHECK all the polymorphic parameters from right appear on left. RIGHT SIDE ONLY
//NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY

    syntax KItem ::= parCheck(K,K) [function] //(T,AlphaMap)
    syntax KItem ::= makeTyVarList(K,K,K) [function] //(TyVars, NewList)
    syntax KItem ::= lengthRet(K,K,K) [function]

    rule <k> checkParUsed
            => parCheck(T,.K) ...</k>
        <tempT> T:K </tempT>

    //rule makeParLists(TList(ListItem(TObject(A:K,ListItem(Arg:KItem) PolyList:List,C:K)) Rest:List),Tlist:List,T
    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,C:K)) Rest:List),.K) => parCheck(TList(ListItem(TObject

    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,ListItem(Par:KItem) ParRest:Lis
        parCheck(TList(ListItem(TObject(ModName,A,B,ListItem(InnerTPiece(C,ParRest,D,E,F)) InnerRest)) Rest),NewS
            requires Par in NewSet

    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,ListItem(Par:KItem) ParRest:Lis
            requires notBool (Par in NewSet)

    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,.List,D:K,E:K,F:K)) InnerRest:
        parCheck(TList(ListItem(TObject(ModName,A,B,InnerRest)) Rest),NewSet)

    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,.List)) Rest:List),NewSet:Set) =>
        parCheck(TList(Rest),NewSet)

    rule parCheck(TList(.List),NewSet:Set) => .K

    rule makeTyVarList('typeVars(A:K,,Rest:K),NewList:List,NewSet:Set) => makeTyVarList(Rest, ListItem(A) NewList,
```

```
    rule makeTyVarList(.TyVars,NewList:List,NewSet:Set) => lengthRet(size(NewList),size(NewSet),NewSet)

    rule lengthRet(A:Int, B:Int, C:K) => C
                requires A ==Int B

    rule lengthRet(A:Int, B:Int, C:K) => error
                requires A =/=Int B

    //rule argSortCheck(TList(ListItem(TObject(A:K,B:K,C:K)

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // STEP 3 Transform T into beta

    syntax KItem ::= "startTTransform"
    syntax KItem ::= "constructDelta"
    syntax KItem ::= "constructBeta"

    rule <k> startTTransform
          => constructDelta
             ~> (constructBeta) ...</k>

    rule <k> constructDelta
          => makeDelta(T,.Map) ...</k>
        <tempT> T:K </tempT>

    syntax KItem ::= makeDelta(K,Map) [function] //(T,Delta)
    syntax KItem ::= newDelta(Map) //Delta
    syntax KItem ::= newBeta(Map) //beta
    syntax List ::= retPolyList(K,List) [function] //(T,Delta)

    rule makeDelta(TList(ListItem(TObject(ModName:K,A:K,Polys:K,C:K)) Rest:List),M:Map) =>
        makeDelta(TList(Rest),M[ModPlusType(ModName,A) <- size(retPolyList(Polys,.List))])
    rule makeDelta(TList(.List),M:Map) => newDelta(M)

    rule retPolyList('typeVars(A:K,,Rest:K),NewList:List) => retPolyList(Rest, ListItem(A) NewList)
    rule retPolyList(.TyVars,L:List) => L

    rule <k> newDelta(M:Map)
          => .K ...</k>
        <tempDelta> OldDelta:K => M </tempDelta>

    rule <k> constructBeta
          => makeBeta(T,.Map) ...</k>
        <tempT> T:K </tempT>

    syntax KItem ::= makeBeta(K,Map) [function] //(T,Beta,Delta)

    rule makeBeta(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(Con:K,H:K,D:K,E:K,F:K)) InnerRest:
        makeBeta(TList(ListItem(TObject(ModName,A,B,InnerRest)) Rest),Beta[ModPlusType(ModName,Con) <- betaParser
    rule makeBeta(TList(ListItem(TObject(ModName:K,A:K,B:K,.List)) Rest:List),Beta:Map) =>
        makeBeta(TList(Rest),Beta)
    rule makeBeta(TList(.List),Beta:Map) =>
        newBeta(Beta)
//    rule makeBeta(TList(ListItem(TObject(ModName:K,A:K,B:K,ListItem(InnerTPiece(C:K,H:K,D:K,E:K,F:K)) InnerRest:
//        makeBeta(TList(ListItem(TObject(ModName,A,B,InnerRest)) Rest),Beta)

    syntax KItem ::= betaParser(K,K,K) [function] //(Tree Piece,NewSyntax,Parameters,Constr)
    syntax Set ::= getTyVarsRHS(K,List) [function]

    syntax KItem ::= forAll(Set,K)
    syntax KItem ::= funtype(K,K)

    syntax Set ::= listToSet(List, Set) [function]

    rule listToSet(ListItem(A:KItem) L:List, S:Set) => listToSet(L, SetItem(A) S)
    rule listToSet(.List, S:Set) => S


//if optbangAtypes, need to see if first variable is a typecon
//if its a typecon then need to go into Delta and see the amount of parameters it has
//then count the number of optbangAtypes after the typecon
    rule betaParser('constrCon(A:K,, B:K), Par:K, Con:K) => forAll(getTyVarsRHS(B,.List), betaParser(B, Par, Con))
```

```
    rule betaParser('optBangATypes('optBangAType('emptyBang(.KList),, 'atypeTyVar(Tyv:K)),, Rest:K), Par:K, Con:K)
    rule betaParser('optBangATypes('optBangAType('emptyBang(.KList),, 'baTypeCon(A:K,, B:K)),, Rest:K), Par:K, Con
    rule betaParser('optBangATypes('optBangAType('emptyBang(.KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K, Con:K
    rule betaParser(.OptBangATypes, Par:K, Con:K) => 'simpleTypeCon(Con,, Par)
//    rule betaParser('optBangATypes('optBangAType('emptyBang(.KList),, 'atypeGTyCon(Tyc:K)),, Rest:KItem)) => getT
//    rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K)) => getTypeVars(Rest)
//    rule getTypeVars('atypeGTyCon(A:K)) => .List
//    rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
//    rule getTypeVars(.OptBangATypes) => .List

    rule getTyVarsRHS(.OptBangATypes,Tylist:List) => listToSet(Tylist, .Set)

    rule <k> newBeta(M:Map)
             => .K ...</k>
         <tempBeta> OldBeta:K => M </tempBeta>

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//    syntax KItem ::= "insertAlphaBetaStar"

    syntax KItem ::= insertABRec(K,List)
    syntax KItem ::= insertAB(K)

    rule <k> insertAlphaBetaStar => insertABRec(Mod, Imp) ...</k>
         <tempModule> Mod:KItem </tempModule>
         <importTree> Imp:List </importTree>

    rule <k> insertABRec(Node:KItem, ListItem(impObject(B:KItem,Node)) I:List) => insertAB(B) ~> insertABRec(Node,

    rule <k> insertABRec(Node:KItem, ListItem(impObject(B:KItem,C:KItem)) I:List) => insertABRec(Node, I) ...</k>
              requires Node =/=K C

    rule <k> insertAB(B) => .K ...</k>

         <tempAlphaStar> Alph:KItem </tempAlphaStar>
         <tempBetaStar> Bet:KItem </tempBetaStar>

         <moduleName> 'moduleName(B) </moduleName>
         <moduleImpAlphas> ImpAlphas:List => ListItem(Alph) ImpAlphas </moduleImpAlphas>
         <moduleImpBetas> ImpBetas:List => ListItem(Bet) ImpBetas </moduleImpBetas>


endmodule
```

CHAPTER 5

MULTIPLE MODULE SUPPORT

The next step is to implement multiple modules into the Haskell semantics. Similar to including files or objects in other programming languages, Haskell modules can include other modules and use functions, types, and typeclasses declared in the module. There are a several considerations and additional checks that need to be made. Modules need to include other modules There cannot be inclusion cycles Modules need to be able to access user defined types from the other modules that are included When referencing types from other modules, there is a scope where if the user makes another type of the same name in the current module then when the user references the type, it refers to the type from the current module. For instance:

```
File 1:
module File1 where
{data A = B
}
File 2:
module File2 where
{data A = B
;data C = B A
}
```

This will compile. The A used in data C is File2.A However, If there are multiple types with the same name declared outside of the current module. If you try to refer to the type without the parent module, there will be a compiler error because there is ambiguity. For instance:

```
File 1:
module File1 where
{data A = B
}
File 2:
module File2 where
{data A = B
}
File 3:
module File3 where
{import File1
;import File2
;data C = B A
}
```

This will not compile because in File 3, type A is ambiguous and can mean File1.A or File2.A Type synonyms need to include polymorphism For instance: The user can write type A a = B a

Since we need to check for module inclusion cycles and also build the set of user defined types for each module and included modules, I decided to use a tree. The plan for the algorithm is as follows 1. Construct tree for module inclusion 2. Check tree for cycles 3. Go to each leaf and recursively go up the tree and build alpha* and beta* for the types of the module and the children and desugar the scope so that each type specifies the scope.

Where alpha is the map of type synonyms declared in the current module and alpha* is the map of type synonyms declared in the current module and all the included modules. Beta is the set of user defined types from using data and newtype declared in the current module. Beta* is the set of user defined types from using data and newtype declared in the current module and all the included modules. Desugar the scope means that when the user references a type, desugar the reference to also include the parent module at all times. The syntax also needed to be changed to allow for multiple modules. The new syntax added is

```
// CUSTOM SYNTAX NOT PART OF OFFICAL HASKELL

  syntax ModuleList ::= Module [klabel('modListSingle)] | Module "<NEXTMODULE>" ModuleList [klabel('modList)]
```

This is because K cannot read mutiple files. So instead all the included modules for a program are dumped into one file and are seperated by the keyword ¡NEXTMODULE¿ This creates a list of modules called ModuleList.

CHAPTER 6

INFERENCING

6.1 Data Structures

The next step is the actual type inferencing algorithm. I needed to create a syntax for polymorphic types that may contain monomorphic type variables and polymorphic type variables. Then I made a map from type constructor names to arities, called Delta. Then I made a map from data constructors and term identifiers to their most general polymorphic types, called beta. The first part was converting the T data structure into beta, which is more suited for type inferencing. Something to note is that for GHC:

$$fx = yx$$

$$yx = fx$$

This set of functions is allowed. When run, the function just simply runs forever. Another thing to note is that the

```
[context =>]
```

part of the syntax for the types is deprecated. https://stackoverflow.com/questions/9345589/guards-vs-if-then-else-vs-cases-in-haskell For functions, function guards, cases, and if-then-else are all equivalent.

A polymorphic data type looks like $\forall abc, (a \to b) \to c$

6.1.1 Inferencing Rules

Haskell is a strong and static type system.

This means that type inferencing can be ran before compilation or running the code. used to ensure that fun

Type inferencing

Haskell's type system is a Hindley-Milner polymorphic type system that has been extended with type classes to account for overloaded function

A type system is a set of rules that assign a property to various constructs in a programming language called type. A type is a property that allows the programmer to add constraints to programs.

6.2 Type theory

Type theory was created by Bertrand Russell to prevent Russell's Paradox for set theory, introduced by Georg Cantor. The issue was that not specifying a certain property for sets allowed sets to contain themselves in Naive Set Theory. So Bertrand Russell prevented this problem by specifying a property called type for objects, and objects cannot contain their own type.

6.3 Lambda Calculus

The Lambda Calculus was created to

6.4 Hindley-Milner

6.5 Definition of Substitution

A substitution is a set of variables and their replacements. Applying a substitution to an expression means to simultaneously replace each variable in the expression with the replacement term.

[http://www.mathcs.duq.edu/simon/Fall04/notes-7-4/node3.html]

6.6 Composition of Substitutions

6.7 Inferencing Algorithm

$$\frac{}{\Gamma \vdash c : \tau \,|\, \mathrm{unify}\{(\tau, \mathrm{freshInstance}(\tau))\}}\;\text{Constant}$$

$$\frac{}{\Gamma \vdash x : \tau \mid \text{unify}\{(\tau, \text{freshInstance}(\Gamma(x)))\}}\ \text{Variable}$$

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2 \mid \sigma}{\Gamma \vdash \backslash\ \text{x} \rightarrow\ \text{e} : \tau \mid \text{unify}\{(\sigma(\tau), \sigma(\tau_1 \rightarrow \tau_2))\} \circ \sigma}\ \text{Lambda}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \mid \sigma_1 \qquad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \mid \sigma_2 \qquad \sigma_2 \circ \sigma_1(\Gamma) \vdash e_3 : \sigma_2 \circ \sigma_1(\tau) \mid \sigma_3}{\Gamma \vdash \text{if}\, e_1\, \text{then}\, e_2\, \text{else}\, e_3 : \tau \mid \sigma_3 \circ \sigma_2 \circ \sigma_1}\ \text{IfThenElse}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \mid \sigma_1 \qquad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau_1) \mid \sigma_2}{\Gamma \vdash e_1 e_2 : \tau \mid \sigma_2 \circ \sigma_1}\ \text{Application}$$

$$\frac{}{\Gamma \vdash c : \tau \mid \text{unify}\{(\tau, \text{freshInstance}(\tau))\}}\ \text{LetIn}$$

[cs 421 gather exp ty substitution mp]

```
requires "haskell-syntax.k"
requires "haskell-configuration.k"
requires "haskell-preprocessing.k"

module HASKELL-TYPE-INFERENCING
    imports HASKELL-SYNTAX
    imports HASKELL-CONFIGURATION
    imports HASKELL-PREPROCESSING

    syntax KItem ::= "Bool" //Boolean

    // STEP 4 Type Inferencing
    syntax KItem ::= inferenceShell(K) [function]//Input, AlphaMap, Beta, Delta, Gamma
    //syntax KItem ::= typeInferenceFun(K,Map,Map,Map,Map,K,K) [function]//Input, Alpha, Beta, Delta, Gamma
    //syntax KItem ::= typeInferenceFun(Map,K,K) //Gamma, Expression, Guessed Type
    syntax Map ::= genGamma(K,Map,K) [function] //Apatlist, Gamma Type
    syntax KItem ::= genLambda(K,K) [function]
    syntax KItem ::= guessType(Int)
//    syntax KItem ::= lambdaReturn(K,K,K)
    syntax KItem ::= freshInstance(K, Int) [function]
    syntax Int ::= paramSize(K) [function]


    syntax KItem ::= mapBag(Map)
    syntax KResult ::= mapBagResult(Map)

    syntax Map ::= gammaSub(Map,Map,Map) [function]//substitution, gamma

    rule <k> performIndividualInferencing => inferenceShell(Code) ...</k>
         <tempModule> Mod:KItem </tempModule>

         <moduleName> 'moduleName(Mod) </moduleName>
         <moduleTempCode> Code:KItem </moduleTempCode>

    rule inferenceShell('topdeclslist('type(A:K,, B:K),, Rest:K)) =>
         inferenceShell(Rest) //constructalpha
    rule inferenceShell('topdeclslist('data(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
    rule inferenceShell('topdeclslist('newtype(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
    rule inferenceShell('topdeclslist('class(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
    rule inferenceShell('topdeclslist('instance(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
    rule inferenceShell('topdeclslist('default(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
    rule inferenceShell('topdeclslist('foreign(A:K,, B:K,, C:K,, D:K),, Rest:K)) =>
         inferenceShell(Rest)
```

```
rule inferenceShell('topdeclslist('topdecldecl(A:K),, Rest:K)) =>
     typeInferenceFun(.ElemList, .Map,A,guessType(0)) ~> inferenceShell(Rest)


rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'declFunLhsRhs(Fn:K,, Lhsrhs:K), Guess:K) =>
     typeInferenceFun(.ElemList, Gamma, Lhsrhs, Guess) ...</k>
rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'eqExpOptDecls(Ex:K,, Optdecls:K), Guess:K) =>
     typeInferenceFun(.ElemList, Gamma, Ex, Guess) ...</k>

//T-App
//rule typeInferenceFun('aexpQVar(Var:K), Alpha:Map, Beta:Map, Delta:Map, (Var |-> Sigma:K) Gamma:Map,.K,.K) =>
//Gamma Proves x:phi(tau) if Gamma(x) = \forall alpha_1, ..., alpha_n . tau
//where phi replaces all occurrences of alpha_1, ..., alpha_n by monotypes tau_1, ..., tau_n

rule <k> typeInferenceFun(.ElemList, (Var |-> Type:K) Gamma:Map, 'aexpQVar(Var:K), Guess:KItem)
      => mapBagResult(uniFun(ListItem(uniPair(Guess,freshInstance(Type, TypeIt))))) ...</k> //Variable rule
     <typeIterator> TypeIt:Int => TypeIt +Int paramSize(Type) </typeIterator>
```

$$\frac{}{\Gamma \vdash c : \tau \,|\, \text{unify}\{(\tau, \text{freshInstance}(\tau))\}}\ \text{Constant}$$

```
rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'aexpGCon('conTyCon(Mid:K,, Gcon:K)), Guess:KItem)
      => mapBagResult(uniFun(ListItem(uniPair(Guess,freshInstance(Type, TypeIt))))) ...</k> //Constant rule
     <tempBeta> (ModPlusType(Mid,Gcon) |-> Type:K) Beta:Map </tempBeta>
     <typeIterator> TypeIt:Int => TypeIt +Int paramSize(Type) </typeIterator>
```

$$\frac{}{\Gamma \vdash x : \tau \,|\, \text{unify}\{(\tau, \text{freshInstance}(\Gamma(x)))\}}\ \text{Variable}$$

```
   syntax KItem ::= typeInferenceFun(ElemList, Map, K, K) [strict(1)]
   syntax KItem ::= typeInferenceFunLambda(ElemList, K, K, K) [strict(1)]
/* automatically generated by the strict(1) in typeInferenceFun or typeInferenceFunAux
   rule typeInferenceFunAux(Es:ElemList, C:K, A:K, B:K) => Es ~> typeInferenceFun(HOLE, C, A, B)
        requires notBool isKResult(Es)
   rule Es:KResult ~> typeInferenceFunAux(HOLE, C:K,A:K, B:K) => typeInferenceFun(Es, C, A, B)
*/

    //lambda rule
    rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'lambdaFun(Apatlist:K,, Ex:K), Guess:KItem)
          => typeInferenceFunLambda(val(typeInferenceFun(.ElemList, genGamma(Apatlist,Gamma,guessType(TypeIt)), gen
         <typeIterator> TypeIt:Int => TypeIt +Int 2 </typeIterator>

    rule <k> typeInferenceFunLambda(valValue(mapBagResult(Sigma:Map)), .ElemList, Tau:K, Tauone:K, Tautwo:K)
         => mapBagResult(compose(uniFun(ListItem(uniPair(typeSub(Sigma,Tau),typeSub(Sigma,funtype(Tauone,Tautwo)))
```

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2 \,|\, \sigma}{\Gamma \vdash \backslash\ \text{x} \rightarrow\ \text{e} : \tau \,|\, \text{unify}\{(\sigma(\tau), \sigma(\tau_1 \rightarrow \tau_2))\} \circ \sigma}\ \text{Lambda}$$

```
    syntax KItem ::= typeInferenceFunAppli(ElemList, Map, K, K, Map) [strict(1)]

    //application rule
    rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'funApp(Eone:K,, Etwo:K), Guess:KItem)
          => typeInferenceFunAppli(val(typeInferenceFun(.ElemList, Gamma, Eone, funtype(guessType(TypeIt),Guess)))
         <typeIterator> TypeIt:Int => TypeIt +Int 1 </typeIterator>

    rule <k> typeInferenceFunAppli(valValue(mapBagResult(Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem, guessTyp
          => typeInferenceFunAppli(val(typeInferenceFun(.ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo, typeSub(S

    rule <k> typeInferenceFunAppli(valValue(mapBagResult(Sigmatwo:Map)), .ElemList, .Map, .K, .K, Sigmaone:Map)
          => mapBagResult(compose(Sigmatwo, Sigmaone)) ...</k>
```

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \,|\, \sigma_1 \qquad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau_1) \,|\, \sigma_2}{\Gamma \vdash e_1 e_2 : \tau \,|\, \sigma_2 \circ \sigma_1}\ \text{Application}$$

```
syntax KItem ::= typeInferenceFunIfThen(ElemList, Map, K, K, K, Map, Map) [strict(1)]

//if_then_else rule
rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'ifThenElse(Eone:K,, Optsem:K,, Etwo:K,, Optsemtwo:K,, Ethree:K
      => typeInferenceFunIfThen(val(typeInferenceFun(.ElemList, Gamma, Eone, Bool)), .ElemList, Gamma, Etwo, E

rule <k> typeInferenceFunIfThen(valValue(mapBagResult(Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem, Ethree
      => typeInferenceFunIfThen(val(typeInferenceFun(.ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo, typeSub(

rule <k> typeInferenceFunIfThen(valValue(mapBagResult(Sigmatwo:Map)), .ElemList, Gamma:Map, .K, Ethree:KItem, 
      => typeInferenceFunIfThen(val(typeInferenceFun(.ElemList, gammaSub(compose(Sigmatwo, Sigmaone), Gamma, .M

rule <k> typeInferenceFunIfThen(valValue(mapBagResult(Sigmathree:Map)), .ElemList, .Map, .K, .K, .K, Sigmaone:M
      => mapBagResult(compose(compose(Sigmathree, Sigmatwo), Sigmaone)) ...</k>
```

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \,|\, \sigma_1 \qquad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \,|\, \sigma_2 \qquad \sigma_2 \circ \sigma_1(\Gamma) \vdash e_3 : \sigma_2 \circ \sigma_1(\tau) \,|\, \sigma_3}{\Gamma \vdash \mathrm{if}\, e_1 \,\mathrm{then}\, e_2 \,\mathrm{else}\, e_3 : \tau \,|\, \sigma_3 \circ \sigma_2 \circ \sigma_1}\ \text{IfThenElse}$$

```
    syntax KItem ::= typeInferenceFunLetIn(ElemList, Map, Map, K, K, K, Int, Int, Map, Map) [strict(1)]
    syntax KItem ::= grabLetDeclName(K, Int) [function]
    syntax KItem ::= grabLetDeclExp(K, Int) [function]
    syntax KItem ::= mapLookup(Map, K) [function]
    syntax Map ::= makeDeclMap(K, Int, Map) [function]
    syntax Map ::= applyGEN(Map, Map, Map, Map) [function]

    //Haskell let in rule (let rec in exp + let in rule combined)
    //gamma |- let rec f1 = e1 and f2 = e2 and f3 = e3 .... in e =>
    //beta, [f1 -> tau1, f2 -> tau2, f3 -> tau3,....] + gamma |- e1 : tau1 | sigma1,  [f1 -> simga1(tau1), f2 -> si
[f1 -> sigma2 o sigma1(tau1), f2 -> sigma2 o sigma1(tau2), f3 -> sigma2 o sigma1(tau3),....] + sigma2 o sigma1(gamm
[f1 -> gen(sigma_n o sigma2 o sigma1(tau1), sigma_n o sigma2 o sigma1(Gamma)), f2 -> gen(tau2), f3 -> gen(tau3),..
    rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'letIn(D:K,, E:K), Guess:KItem)
          => typeInferenceFunLetIn(.ElemList, Gamma, makeDeclMap(D, TypeIt, .Map), D, E, Guess, 0, TypeIt, .Map, B
         <typeIterator> TypeIt:Int => TypeIt +Int size(makeDeclMap(D, TypeIt, .Map)) </typeIterator>
         <tempBeta> Beta:Map </tempBeta>

    rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap:Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, Typ
           => typeInferenceFunLetIn(val(typeInferenceFun(.ElemList, Gamma DeclMap, grabLetDeclExp(D, Iter), mapLook
          //=> typeInferenceFunLetIn(val(typeInferenceFun(DeclMap, grabLetDeclExp(D, Iter +Int TypeIt), Guess)), .M
          requires Iter <Int (size(DeclMap))

    rule <k> typeInferenceFunLetIn(valValue(mapBagResult(Sigma:Map)), .ElemList, Gamma:Map, DeclMap:Map, D:KItem, E
          => typeInferenceFunLetIn(.ElemList, gammaSub(Sigma,Gamma,.Map), gammaSub(Sigma, DeclMap,.Map), D, E, type
          requires Iter <Int (size(DeclMap))

    rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap:Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, Typ
          => typeInferenceFunLetIn(val(typeInferenceFun(.ElemList, Gamma applyGEN(Gamma, DeclMap, .Map, Beta), E, G
          requires Iter >=Int (size(DeclMap))

    rule <k> typeInferenceFunLetIn(valValue(mapBagResult(Sigma:Map)), .ElemList, Gamma:Map, DeclMap:Map, D:KItem, E
          => mapBagResult(compose(Sigma, OldSigma))...</k>
          requires Iter >=Int (size(DeclMap))
```

$$\frac{}{\Gamma \vdash c : \tau \,|\, \mathrm{unify}\{(\tau, \mathrm{freshInstance}(\tau))\}}\ \text{LetIn}$$

```
rule mapLookup((Name |-> Type:KItem) DeclMap:Map, Name:KItem) => Type
rule mapLookup(DeclMap:Map, Name:KItem) => Name
     requires notBool(Name in keys(DeclMap))

rule makeDeclMap('decls(Dec:K), TypeIt:Int, NewMap:Map) => makeDeclMap(Dec, TypeIt, NewMap)
rule makeDeclMap('declsList('declPatRhs('apatVar(Var:K),, Righthand:K),, Rest:K), TypeIt:Int, NewMap:Map) => m
rule makeDeclMap(.DeclsList, TypeIt:Int, NewMap:Map) => NewMap

rule grabLetDeclName('decls(Dec:K), Iter:Int) => grabLetDeclName(Dec, Iter)
rule grabLetDeclName('declsList(Dec:K,, Rest:K), Iter:Int) => grabLetDeclName(Rest, Iter -Int 1)
     requires Iter >Int 0
rule grabLetDeclName('declsList('declPatRhs('apatVar(Var:K),, Righthand:K),, Rest:K), Iter:Int) => Var
     requires Iter <=Int 0


rule grabLetDeclExp('decls(Dec:K), Iter:Int) => grabLetDeclExp(Dec, Iter)
rule grabLetDeclExp('declsList(Dec:K,, Rest:K), Iter:Int) => grabLetDeclExp(Rest, Iter -Int 1)
     requires Iter >Int 0
rule grabLetDeclExp('declsList('declPatRhs('apatVar(Var:K),, Righthand:K),, Rest:K), Iter:Int) => grabLetDeclEx
     requires Iter <=Int 0
rule grabLetDeclExp('eqExpOptDecls(Righthand:K,, Opt:K), Iter:Int) => 'eqExpOptDecls(Righthand,, Opt)

rule genGamma('apatVar(Vari:K), Gamma:Map, Guess:K) => Gamma[Vari <- Guess]
rule genGamma('apatCon(Vari:K,, Pattwo:K), Gamma:Map, Guess:K) => Gamma[Vari <- Guess]

rule genLambda('apatVar(Vari:K), Ex:K) => Ex
rule genLambda('apatCon(Vari:K,, Pattwo:K), Ex:K) => 'lambdaFun(Pattwo,, Ex)


rule gammaSub(Sigma:Map, (Key:KItem |-> Type:KItem) Gamma:Map, Newgamma:Map)
    => gammaSub(Sigma, Gamma, Newgamma[Key <- typeSub(Sigma, Type) ] )

rule gammaSub(Sigma:Map, .Map, Newgamma:Map)
  => Newgamma

rule freshInstance(guessType(TypeIt:Int), Iter:Int) => guessType(TypeIt)
rule freshInstance(forAll(.Set, B:K), Iter:Int) => B
rule freshInstance(forAll(SetItem(C:KItem) A:Set, B:K), Iter:Int) => freshInstance(forAll(A, freshInstanceInner

syntax KItem ::= freshInstanceInner(K,K,Int) [function]

rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K), Iter:Int) => funtype(freshInstanceInner(Repl,A,Iter),fre
rule freshInstanceInner(Repl:KItem, Repl, Iter:Int) => guessType(Iter)
rule freshInstanceInner(Repl:KItem, Target:KItem, Iter:Int) => Target [owise]

rule paramSize(forAll(A:Set, B:K)) => size(A)
rule paramSize(A:K) => 0 [owise]


 rule applyGEN(Gamma:Map, (Key:KItem |-> Type:KItem) DeclMap:Map, NewMap:Map, Beta:Map)
   => applyGEN(Gamma, DeclMap, NewMap[Key <- gen(Gamma, Type, Beta)], Beta)

 rule applyGEN(Gamma:Map, .Map, NewMap:Map, Beta:Map)
   => NewMap

//GEN
//GEN(Gamma, Tau) => Forall alpha

syntax KItem ::= gen(Map, K, Map) [function]
syntax Set ::= freeVarsTy(K, Map) [function]
syntax Set ::= freeVarsEnv(Map, Map) [function]


rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map) => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -
rule gen(Gamma:Map, Tau:KItem, Beta:Map) => forAll(freeVarsTy(Tau, Beta) -Set freeVarsEnv(Gamma, Beta), Tau) [

//rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map) => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta)

rule freeVarsTy(guessType(TypeIt:Int), Beta:Map) => SetItem(guessType(TypeIt:Int))
rule freeVarsTy(funtype(Tauone:KItem, Tautwo:KItem), Beta:Map) => freeVarsTy(Tauone, Beta) freeVarsTy(Tautwo,
rule freeVarsTy(Tau:KItem, Beta:Map) => .Set
     requires (forAll(.Set, Tau)) in values(Beta)
```

```
rule freeVarsTy(forAll(Para:Set, Tau:KItem), Beta:Map) => freeVarsTy(Tau, Beta) -Set Para
rule freeVarsEnv(Gamma:Map, Beta:Map) => listToSet(values(Beta), .Set)
```

6.7.1 Unification Algorithm

Let $S = \{(s_1, t_1), (s_2, t_2), \cdots, (s_n, t_n)\}$ be a unification problem.

Case $S = \{\}$: Unif(S) = Identity function (ie no substitution)

Case $S = (s, t) \cup S'$): Four main steps

Delete: if $s = t$ (they are the same term) then $Unif(S) = Unif(S')$

Decompose: if $s = f(q_1, \cdots, q_m)$ and $t = f(r_1, \cdots, r_m)$ (same f, same m!), then Unif(S) = Unif($\{(q_1, r_1), ..., (q_m, r_m)\} \cup S'$)

Orient: if $t = x$ is a variable, and s is not a variable, Unif(S) = Unif($\{(x, s)\} \cup$ S')

Eliminate: if s = x is a variable, and x does not occur in t (the occurs check), then

Let $\phi = x \mapsto t$

Let ψ = Unif($\phi(S')$)

Unif(S) = $\{x \mapsto \psi(t)\} \circ \psi$

Note: x $\longrightarrow$ a o y $\longrightarrow$ b = y $\longrightarrow$ (x $\longrightarrow$ a(b) o x $\longrightarrow$ a if y not in a

[cs 421 class notes]

```
//Unification

syntax Map ::= uniFun(List) [function]
syntax Bool ::= isVarType(K) [function]
syntax Bool ::= notChildVar(K,K) [function]
syntax KItem ::= uniPair(K,K)

syntax List ::= uniSub(Map,K) [function] //apply substitution to unification

syntax KItem ::= typeSub(Map,K) [function] //apply substitution to type
syntax Map ::= compose(Map,Map) [function]

rule uniFun(.List) => .Map //substi(.K,.K) is id substitution

rule uniFun(ListItem(uniPair(S:K,S)) Rest:List) => uniFun(Rest)  //delete rule

rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List) => uniFun(ListItem(uniPair(T,S)) Rest) //orient rule
     requires isVarType(T) andBool (notBool isVarType(S))

rule uniFun(ListItem(uniPair(funtype(A:K, B:K), funtype(C:K, D:K))) Rest:List) => uniFun(ListItem(uniPair(A, C

rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List)
  => compose((S |-> typeSub(uniFun(uniSub((S |-> T),Rest)),T)),uniFun(uniSub((S |-> T),Rest))) //eliminate rule
//  => compose(uniFun(uniSub((S |-> T),Rest)),(S |-> typeSub(uniFun(uniSub((S |-> T),Rest)),T))) //eliminate ru
     requires isVarType(S) andBool notChildVar(S,T)

rule isVarType(S:K) => true
     requires getKLabel(S) ==KLabel 'guessType
rule isVarType(S:K) => false [owise]

rule notChildVar(S:K,T:K) => true
```

```
    rule uniSub(Sigma:Map,.List) => .List
    rule uniSub(.Map,L:List) => L
    rule uniSub(Sigma:Map, Rest:List ListItem(uniPair(A:K, B:K))) => uniSub(Sigma, Rest) ListItem(uniPair(typeSub(S

    //rule typeSub(substi(.Map),Tau:KItem) => Tau
    rule typeSub(Sigma:Map (Tau |-> Newtau:KItem),Tau:KItem) => typeSub(Sigma (Tau |-> Newtau),Newtau)
    rule typeSub(Sigma:Map,funtype(Tauone:KItem,Tautwo:KItem)) => funtype(typeSub(Sigma,Tauone),typeSub(Sigma,Tautw
    rule typeSub(Sigma:Map,Tau:KItem) => Tau [owise]

    syntax Map ::= composeIn(Map, Map, Map, K, K) [function]

    rule compose(Sigmaone:Map, Sigmatwo:Map) => composeIn(Sigmaone, Sigmatwo, .Map, .K, .K)

    rule composeIn(Sigmaone:Map, (Key:KItem |-> Type:KItem) Sigmatwo:Map, NewMap:Map, .K, .K) => composeIn(Sigmaone

    rule composeIn((Keyone |-> Typetwo:KItem) Sigmaone:Map, Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem)

    rule composeIn((Typeone |-> Typetwo:KItem) Sigmaone:Map, Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem)
         requires notBool(Keyone in keys(Sigmaone))

    rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone, Sign

    rule composeIn(Sigmaone:Map, .Map, NewMap:Map, .K, .K) => Sigmaone NewMap
endmodule
```

CHAPTER 7

CONCLUSION

This project helped me with my K knowledge. I also learned a lot about how compilers work and how programming languages are defined. This also was a great test of the K-Framework to define an executable formal semantics for a real programming language.

REFERENCES