# HASKELL SYNTAX AND STATIC SEMANTICS WRITTEN IN K-FRAMEWORK

BY

BRADLEY MORRELL

THESIS

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

Urbana, Illinois

Adviser:

Professor Elsa Gunter

# Abstract

This thesis introduces 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. Many layers 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, not 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 a separate file. 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. Due to being written in K, the semantics introduced here is mathematically precise and executable. Since the semantics is executable, the semantics can be tested against test sets to validate the correctness of the semantics. The executability of the semantics was utilized 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

Chapter 1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

Chapter 2 Context Free Syntax . . . . . . . . . . . . . . . . . . . . . . . . 3
2.1 K Explanation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.2 Syntax Explanation . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.3 Implementation of Section 10.2 . . . . . . . . . . . . . . . . . . . . 5
2.4 Implementation of Section 10.5 . . . . . . . . . . . . . . . . . . . . 6

Chapter 3 Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.1 Alpha . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Beta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.3 Delta . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.4 Import Data Structure . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.5 Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

Chapter 4 Context-Sensitive Checks . . . . . . . . . . . . . . . . . . . . . 17
4.1 Data Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2 Datatype Constructors . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3 Initial Data Structures . . . . . . . . . . . . . . . . . . . . . . . . 19
4.4 context-sensitive Checks . . . . . . . . . . . . . . . . . . . . . . . 23

Chapter 5 Inferencing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.1 Type theory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.2 Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.3 Definition of Substitution . . . . . . . . . . . . . . . . . . . . . . 38
5.4 Inference Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.5 Mutual Recursion . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.6 Helper Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.7 Fresh Instance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.8 GEN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.9 Unification Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.10 Composition of Substitutions . . . . . . . . . . . . . . . . . . . . . 55
5.11 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

Chapter 6 Multiple Module Support . . . . . . . . . . . . . . . . . . . . . 59
6.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

Chapter 7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

Appendix A haskell-syntax.k . . . . . . . . . . . . . . . . . . . . . . . 62

Appendix B haskell-configuration.k . . . . . . . . . . . . . . . . . . . . 74

Appendix C haskell-preprocessing.k . . . . . . . . . . . . . . . . . . . . 76

Appendix D haskell-type-inferencing.k . . . . . . . . . . . . . . . . . . . 97

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

# Chapter 1

# Introduction

One of the inherent problems of engineered systems in general is that the design of the system is not proven to work. 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 issues introduced by implementing the design incorrectly.

Within the context of computer programs, 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 the correctness of a system. Without formal methods, the only way to reason about a system is by testing it against various 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 [1, 2] 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 run 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, which is 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 dependent on the function input. Strong static typing means that before a program is run, a type inference algorithm infers the type of the program and ensures that all functions and function applications are allowed with regard to the type of the inputs and outputs. "Static" refers to the fact that type inference is performed before the code is run, 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.

The Haskell 2010 report gives a cursory description of the type system as a Standard Hindley Milner polymorphic type system, but gives no further indication of how this applies to the specifics of the Haskell syntax.

In addition, by studying the standard Haskell compiler, one can see that there must be additional context-sensitive checks that must be made prior to type inference.

In Robin Milner's work, their type system is based on a more simple ML system. In Haskell, the semantics needs to worry about parsing, preprocessing, modules, expressions, declarations, and so on.

Also, an important part of the Haskell language is mutual recursion of functions. Functions can refer to each other within a Haskell program.

In the specification in this paper, a more complete specification of parts of the type system is presented as a family of mutually inductive rules. Such a presentation is not only the basis of an executable semantics but can later provide the basis of formal and rigorous proofs.

This project details the syntax of Haskell and the type system of Haskell in K.

The repository of the K semantics is provided at:

`https://github.com/liyili2/haskell-semantics/`

# Chapter 2

# Context Free Syntax

This chapter details the first part of the static semantics. In order for any context-sensitive checks or type inference to be done, the test programs first need to be parsed into an abstract syntax tree. The context-sensitive checks and type inference can then be performed upon the tree. A difficulty with implementing a grammar into K is that the grammar originally is written in sort descending order in a document. The goal was to build a grammar that can parse actual programs and ensure there were no bugs. To do this, I started with small example programs, wrote out the example abstract syntax tree, and included the sorts necessary to parse them. Then if they did not parse correctly I could debug. I then wrote bigger and bigger example programs and included more and more sorts until all the grammar was included.

The Haskell 2010 report [3] 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.

Section 10.1 [3] specifies the notation used in the grammar.

The notations of 10.1 are always in bold in the grammar. The notation is the same as standard syntax definition notation. So an example production in the document looks like

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

This means that

```
modid .
```

is optional, and the brackets are not terminals, but the period is a terminal. Any symbol that is not in bold needs to be written in the program in order to parse correctly.

## 2.1 K Explanation

### 2.1.1 K Labels

In K, semantic rules are called K rules. The tag

```
[klabel('exampleLabel)]
```

means that in the abstract syntax tree created by K, a term can be referred to using that `klabel` in a K rule.

### 2.1.2 Strict

The tag

```
[strict]
```

means that K will add two additional heating and cooling rules. This means the child of the `KItem` will be placed in front of the `KItem` to be evaluated first until it is given the `KResult` label. Then it is placed back into the `KItem`.

## 2.2 Syntax Explanation

Many parts of the grammar posed challenges when implementing in K. For instance, a sort definition that includes an option could be just written using a pipe in the K syntax.

So the example production

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

is written in the K syntax as split into two options.

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

However, an issue arises when you have something written on the right-hand side of the production like

`data [ context => ] simpletype [ = constrs ] [deriving]`
for the `topdecl` sort. If each optional sort were split into two options, then a production that includes $n$ optional sorts would require $2^n$ options. This would create an unnecessarily large syntax in K. It also would quickly become hard to read or maintain.

Instead of having many optional sorts in a production similar to the original grammar, the K syntax instead utilizes an entirely new sort which can optionally include just an empty string.

For instance, an optional sort in the original grammar could look like:

`[ context => ]`

The corresponding production in the K syntax instead includes the sort called `OptContext`, where OptContext is defined as

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

and so the right hand side of the previous production would now look like

```
"data" OptContext SimpleType OptConstrs OptDeriving  [klabel
   ('data)]
```

This is acceptable because Haskell is not an order sorted algebra, so introducing new sorts that are not originally in the grammar is allowed.

## 2.3 Implementation of Section 10.2

The first section of the syntax introduces the syntax for the keywords, constants, special symbols, and variables that comprise the terminals for the remaining context-free grammar as presented in section 10.2 of the 2010 Haskell report. The rest of section 10.2 is included in Appendix A.

Variable IDs start with a lowercase letter and Constructor IDs start with an uppercase letter.

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

```
syntax ConId ::= Token{[A-Z][a-zA-Z \_0-9\']*} [
   onlyLabel]
```

## 2.4 Implementation of Section 10.5

The following introduces the sorts of the context-free grammar of the Haskell extended syntax. The rest of section 10.5 is included in Appendix A.

### 2.4.1 Modules

We start with modules.

```
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)]
```

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.

### 2.4.2 ImpDecls

An `ImpDecl` is an import declaration. An import is another module that this module depends on. The definition of `ImpDecl` is the following:

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

The following example program has a Module with a name and a body of only `ImpDecls`. It has one `ImpDecl`.

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

The following example program has a Module with no name and and a body of only `ImpDecls`. It has one `ImpDecl`.

```
import Bar
```

Another example program is

```
module Simp1 where
{import Test
}
```

The corresponding abstract syntax tree in K is

```
'module`('moduleName`(#token("Simp1","ConId")), '
   bodyimpdecls`('impDecls`('impDecl`('
   emptyQualified`( .::KList), #token("Test","ConId
   "), 'emptyOptAsModId`(.::KList), '
   emptyOptImpSpec`(.::KList)), '.List{"'impDecls
   "}`(.::KList))))
```

The `module` structure contains two children. The first child of `module` is `moduleName` which contains the token `Simp1`. `Simp1` is a constructor ID because it starts with a capital letter. The second child of `module` is `bodyimpdecls`. This contains the sort `impDecls` which is a list of `impDecls`. This program has only one `impDecl`. Since there is no `qualified`, the `impDecl` contains the child `emptyQualified`, followed by the token `Test`. Since there is no `AsModId` or `ImpSpec`, the last two children are `emptyOptAsModId` and `emptyOptImpSpec`.

### 2.4.3 TopDecls

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.

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.

```
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)]
```

### 2.4.4 Decls

A `Decl` is any general declaration. So something like

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

is a `Decl`.

The following is the definition of `Decl` and related sorts.

```
        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)]
```

This section introduces the sorts whose parent is `Decl`. This contains general declarations, function left-hand side and right-hand side, function guards, and expressions.

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

        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
```

### 2.4.5 LExp

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

The different LExp types are a lambda expression, a let-in expression, an if statement, a case statement, and a do block.

```
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)]
```

### 2.4.6 AExp

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

QVar is a qualified variable and GCon is a general constructor.

```
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 "}"
> "(" Exp ")"             [bracket]
| "(" ExpTuple ")"
| "[" ExpList "]"
| "[" Exp OptExpComma ".." OptExp "]"
| "[" Exp "|" Quals "]"
| "(" InfixExp QOp ")"
| "(" QOp InfixExp ")"
```

# Chapter 3

# Configuration

K is a framework that is used to define formal specifications of programming languages. The K framework is designed to easily correspond to a finite automaton. In a different programming language, it is much harder to represent the program as a formal model because it would require modeling many different parts of the machine. K, on the other hand, can directly compile to Isabelle proofs and be verified. The rules in K are already mathematically precise due to being written in K.

However, code written in K is also executable. It can run example programs using the formal semantics written in it. This means that the K code can run test sets for validation. K code can be considered validated if it passes all the standard test suites provided for a compiler or an interpreter, for example.

A configuration of an automaton is the definition of the specific structure that contains all code and memory that the automaton contains and operates on.

K is used for defining a state machine and the K rules define the transition rules for the state machine. The configuration of the state machine is made up of K cells. The K cells contain the syntax data structure representing the code of the example program. They also contain the memory of the state machine. An actual state of the state machine in K is when the cells each have some term inside of them.

Syntax for a programming language can be written in K using the constructor `syntax` while semantic rules are written in K using the constructor `rule`.

K rules can be used to edit the program code as well as edit the other cells that make up the program state.

In the description below I give incrementally the various components that make up the configuration used in the Haskell static semantics given in this

paper, along with a brief description of what they name and examples of what they hold and how they are used.
The cells are defined using XML syntax.

```
requires "haskell-syntax.k"

module HASKELL-CONFIGURATION
    imports HASKELL-SYNTAX

    syntax KItem ::= "startImportRecursion"
    syntax KItem ::= callInit(K)

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

The $< k >$ cell is the cell that computation takes place in. The abstract syntax tree is initially placed into the $< k >$ cell.
The command

```
$PGM:ModuleList
```

means that the parsed tree appears in this cell and the sort that contains all other sorts is `ModuleList`.
Putting $.K$ means that the cell is initially empty.
The name of the current module is `tempModule`. The current code is `tempCode`.
The cell `typeIterator` is used for creating a fresh type variable for the inference algorithm. It has the current count of how many fresh type variables were created.

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

## 3.1 Alpha

The data structure `Alpha` is a map of type renamings. More is explained in Chapter 5.

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 `MyBooltwo` is an alias for `MyBool` and `MyBooltwo` $->$ `MyBool` in the map. However, we want to check and reject programs that have the same name as an alias for multiple types, so the semantics does not initially use a K Map which has idempotence. However, once it makes this check, it then switches to using a K Map. This is what `tempAlphaMap` is.

`.Map` means that the cell starts with an empty map.

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

## 3.2 Beta

`tempT` contains all user-defined data types. `tempT` is organized in such a way that makes context-sensitive checks easy to perform. `tempBeta` contains all user-defined data types organized so that type inference is easy to perform. More is explained in Chapter 5.

```
<tempBeta> .Map </tempBeta>
<tempT> .K </tempT>
```

### 3.2.1 Example

If the user makes the data type `CusBool` in module `Simp5`, and declares it with this example:

```
data CusBool = True2 | False2
```

then the corresponding `tempBeta` should look as below. Note how the monomorphic datatype just has an empty `forAll` and `TyVars`. For more on monomorphic and polymorphic types, see Chapter 5.

```
<tempBeta>
    ModPlusType ( Simp5 , False2 ) |-> forAll ( .Set ,
        CusBool .TyVars )
```

```
            ModPlusType ( Simp5 , True2 ) |-> forAll ( .Set ,
               CusBool .TyVars )
        </tempBeta>
```

If the user makes the data type `CusBool` in module `Simp5`, and declares it with the example

```
    data CusBool a b = True2 a | False2 b
```

then the corresponding `tempBeta` should look like this:

```
        <tempBeta>
            ModPlusType ( Simp5 , False2 ) |-> forAll ( a b ,
               funtype ( b , CusBool a b ) )
            ModPlusType ( Simp5 , True2 ) |-> forAll ( a b ,
               funtype ( a , CusBool a b ) )
        </tempBeta>
```

## 3.3 Delta

`tempDelta` contains the arity of the user-defined data types. So if a user-defined data type takes in two parameters, `tempDelta` will contain a mapping from the module and the data type name to the number 2. Again, it is initially empty.

```
                <tempDelta> .Map </tempDelta>
```

### 3.3.1 Example

If the user makes the data type `CusBool` in module `Simp5`, and declares it with the example

```
    data CusBool a b = True2 a | False2 b
```

then the corresponding `tempDelta` should look like this:

```
    <tempDelta>
         ModPlusType ( Simp5 , CusBool ) |-> 2
    </tempDelta>
```

## 3.4 Import Data Structure

`importTree`, `recurImportTree`, and `impTreeVMap` contain the data necessary for the directed acyclic graph representing imports.

```
<importTree> .List </importTree>
<recurImportTree> .List </recurImportTree>
<impTreeVMap> .Map </impTreeVMap>
```

More is explained in Chapter 6.

## 3.5 Modules

The modules cell contains all modules that were checked and inferred already. Multiplicity means that there can be multiple module cells.

```
                <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
```

The components here are the module specific versions of the ones just discussed.

# Chapter 4

# Context-Sensitive Checks

The standard Haskell Compiler is called the Glasgow Haskell Compiler [4]. It is otherwise known as GHC. By testing GHC, it is apparent that the compiler will check and reject programs that are syntactically valid, but have additional issues that make the programs invalid. This means that GHC does in fact make additional context-sensitive checks that are not part of the context free grammar. In the semantics in this paper, there are also context-sensitive checks that are made prior to type inference to reject programs that GHC rejects.

## 4.1 Data Types

Within the context of a computer program, a data type is a property of data that tells the compiler or interpreter more about how the data is supposed to be used within the context of the program. This is useful for having the compiler find bugs that occur from the programmer misusing data [5]. In Haskell, a user can create a custom data type. This is referred to as a user-defined type. Then when the user creates functions or any other expression, they can operate on their own data type.

### 4.1.1 Polymorphim

A polymorphic data type is a data type that can generalize over other data types. A monomorphic data type is a data type that does not do this.

## 4.2 Datatype Constructors

In order to perform context-sensitive checks to make sure that the user did not have errors when creating types, the types are placed into data structures that make context-sensitive checks easy to perform.

Section 4 of the Haskell 2010 report [3] specifies the Haskell type system.

In the `TopDecl` sort, there are three type constructors that are used to create user-defined data types. These are `data`, `type`, and `newtype`.

```
syntax TopDecl ::= Decl [klabel('topdecldecl)]
                 > "type" SimpleType "=" Type [klabel('
                    type)]
                 | "data" OptContext SimpleType
                    OptConstrs OptDeriving  [klabel('
                    data)]
                 | "newtype" OptContext SimpleType "="
                    NewConstr OptDeriving [klabel('
                    newtype)]
```

These three type constructors are used to create user-defined types.

### 4.2.1 Type

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.

### 4.2.2 Data

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 type constructor `Date` followed by three integers.

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

This is a new polymorphic type with polymorphic parameter a, that has the type constructor *Number*.

### 4.2.3 Newtype

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.

### 4.2.4 Context

Another thing to note is that the

```
[context =>]
```

part of the syntax for the types is deprecated [6].

## 4.3 Initial Data Structures

The semantics includes a map, called alpha, of new type names as the keys and their declared types as the entries. It collects all appearances of the type constructor `type` in the program, and puts `simpletype` $\mapsto$ `type` in the alpha map. However, one of the checks that is made, is whether a user declared multiple definitions with texttttype, but maps in K only allow for unique keys. So initially the semantics creates a set of tuples, and after checking for multiple type declarations, it then changes the alpha set to a map.

The second data structure that the semantics creates 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,K) //(module name, 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)
```

The structure, `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. The semantics then uses these data structures to perform these checks, and afterwards uses different data structures to perform type inference.

### 4.3.1 Example T

If the user makes the data type `CusBool` in module `Simp5`, and declares it with this example

```
data CusBool = True2 | False2
```

then the corresponding `tempT` should look like this.

```
<tempT>
    TList ( ListItem ( TObject ( Simp5 , CusBool , .
        TyVars ,
         ListItem ( InnerTPiece ( True2 , .List , .List ,
              True2
           .OptBangATypes , CusBool ) )
         ListItem ( InnerTPiece ( False2 , .List , .List
            , False2
           .OptBangATypes , CusBool ) ) ) ) )
</tempT>
```

### 4.3.2 K Code

The following parses the tree and searches for the user-defined types to place into the data structures.

```
//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
    pieces)
syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type
   constructor, poly type vars, argument sorts, entire
   constr block, type name)

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 <k> fetchTypes('simpleTypeCon(I:TyCon,, H:TyVars),
   'atypeGTyCon(C:K), Rest:K, preModule(AList(M:List),
   L:K)) => getPreModule(Rest,preModule(AList(ListItem(
   AObject(ModPlusType(ModName,I),C)) M), 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))) => getPreModule(Rest,preModule(AList(M),
   TList(ListItem(TObject(ModName,I,H,makeInnerT(I,H,D)
   )) ListInside))) ...</k>
     <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),C,A)) makeInnerT(A,B,D)

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('constrCon(A:K,, B:K)) => A

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
```

## 4.4 context-sensitive Checks

For the context-sensitive checks, the semantics performs several checks that are not inclusive to all context-sensitive checks that need to be made in a complete Haskell semantics, but enough for some sanity of the type inference function.

### K Code

The following code introduces all checks that need to be made.

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

### 4.4.1 Name Collision

The programmer should not be able to make two user-defined datatypes with the same name, even if one is created using the constructor `data` and another is created using the constructor `type` for instance.

The following example should not be allowed.

```
data Date = Date Int
;type Date = Contwo Int
```

K Code

The following is the code for the check.

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

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

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

### 4.4.2 Type Constructor Collision

The programmer should not be able to use the same type constructors when making different options for their types or use the same type constructors for different types.

The following example should not be allowed.

```
data Date = Date Int | Date Bool
```

The following example should also not be allowed.

```
data Date = Date Int
;data Datetwo = Date Int
```

K Code

The following is the code for the check.

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

### 4.4.3 Alpha Cycle Check

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`.

The following example should not be allowed.

```
type Birthday = Date
;type Date = Birthday
```

The following example should also not be allowed if Date is not defined anywhere.

```
type Birthday = Date
```

## K Code

The following is the code for the check.

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

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,
   ListItem(B) ListItem(A) D, SetItem(B) SetItem(A) G,
   B, .K,T,S)
rule aloopCheck(Alpha:Map (H |-> B:KItem), D:List, G:Set
   , H:KItem, .K,T:K,S:Set) => aloopCheck(Alpha,
   ListItem(B) D, SetItem(B) G, B, .K,T,S)

rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:
   K,S:Set) => aloopCheck(Alpha, .List, .Set, .K,
   lengthCheck(size(G),size(D)),T,G S) //type rename
   loop ERROR
     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 ERROR
```

```
         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(A) D)
    rule typeSet(TList(.List), D:Set) => D

    rule aloopCheck(.Map, .List, .Set, .K, .K,T:K, S:Set) =>
        .K
```

### 4.4.4 Argument Sort Check

The argument sorts for types defined using the `data` keyword or the `newtype` keyword should be types that exist.

The following example should not be allowed if `Date` is not defined anywhere.

```
Data Birthday = Birthday Date
```

K Code

The following is the code for the check.

```
//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)
        ArgsRest:List,E:K,F:K)) InnerRest:List)) TListRest:
       List),AlphaMap:Map,Tset:Set) => argSortCheck(TList(
       ListItem(TObject(ModName,A,B,ListItem(InnerTPiece(C,
       D,ArgsRest,E,F)) InnerRest)) TListRest),AlphaMap,
       Tset)
         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)
       ArgsRest:List,E:K,F:K)) InnerRest:List)) TListRest:
       List),AlphaMap:Map,Tset:Set) => error
         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))
       InnerRest:List)) TListRest:List),AlphaMap:Map,Tset:
       Set) => argSortCheck(TList(ListItem(TObject(ModName,
       A,B,InnerRest)) TListRest),AlphaMap,Tset)

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

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

### 4.4.5 Polymorphic Parameter Check

The polymorphic parameters that appear on the right-hand side of a `data` declaration need to appear on the left-hand side as well.

The following example should not be allowed.

```
Data Newtype = New a b
```

Also, The polymorphic parameters that appear on the left-hand side of a `data` declaration need to be unique. However, the parameters that appear on the right-hand side do not need to be unique.

The following example should not be allowed.

```
Data Newtype a a = New a
```

However, the following example should be allowed.

```
Data Newtype a = New a a
```

## K Code

The following is the code for the check.

```
//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 parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,C
       :K)) Rest:List),.K) => parCheck(TList(ListItem(
       TObject(ModName,A:K,B:K,C:K)) Rest:List),
       makeTyVarList(B,.List,.Set))

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

    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,
       ListItem(InnerTPiece(C:K,ListItem(Par:KItem) ParRest
       :List,D:K,E:K,F:K)) InnerRest:List)) Rest:List),
       NewSet:Set) => error
             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:List)) Rest:List),NewSet:Set) =>
```

```
     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, SetItem(A) NewSet)

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
```

# Chapter 5

# Inferencing

Once context-sensitive checks are made, the module can now be type inferred. Type inference is used to find the most general type of an expression. This is useful for assigning the most general type to variables and functions. It also is useful for performing type checking if the programmer specifies what type a function should take in or what a variable should be. 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 [7]. The Haskell 2010 report says that Haskell's type system is a Hindley-Milner polymorphic type system [8, 9] that has been extended with type classes to account for overloaded functions. The semantics in this paper makes precise what this means with regard to Haskell's specific syntax. The type inference function infers the type of the expression that is fed into it. A purpose of it is to ensure that all functions and function applications are allowed with regard to the types of the inputs and outputs.

## 5.1 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 [10]. Type theory turns out to be useful for computer science.

### 5.1.1 Simply Typed Lambda Calculus

The Lambda Calculus was created by Alonzo Church as part of his research in foundations of mathematics. However, the Lambda Calculus was logically inconsistent, similar to set theory, shown by the Kleene-Rosser paradox. In order to prevent the Kleene-Rosser paradox for the Lambda Calculus, Church introduced the concept of types into the Lambda Calculus. This is the Simply Typed Lambda Calculus [11]. Later, it was shown that there is a correspondence between programming languages and the Lambda Calculus, which makes the Lambda Calculus a potential abstraction for programming languages. Later still, it was shown that there is a correspondence between proof calculi and type systems, which links mathematics between formal logic and programs. This is the Curry-Howard correspondence.

### 5.1.2 System F

System F [12] introduced the concept of polymorphic types into the Lambda Calculus. It accomplishes this by using universal quantifiers over the types in System F.

### 5.1.3 Hindley-Milner

The concept of type systems grew out of further complications to the types used in the Lambda Calculus. The Hindley-Milner type system adds the concept of type inference to the Lambda Calculus. Type inference used in Hindley-Milner can acquire the most general type of a program without the user giving any type annotations.

### 5.1.4 Meta Language

The theoretical research in type theory and programming language semantics eventually lead to the Meta Language, also known as ML. It utilizes the Hindley-Milner type system. It also features a complete formal specification.

## 5.2 Data Structures

A monomorphic data type has no quantifiers. It is either a ground term itself, or a term composed of ground terms. For instance, a monomorphic data type can be $(a \to b) \to c$ if a, b, and c are also types.

In the K semantics, the equivalent data type looks like

```
forAll ( .Set , funtype ( funtype ( a , b ) , c ) )
```

Note how there are no quantified variables.

A polymorphic data type, on the other hand, has quantifiers. An example polymorphic data type could look like $\forall abc\,.\,(a \to b) \to c$

In the K semantics, the equivalent data type looks like

```
forAll ( a b c , funtype ( funtype ( a , b ) , c ) )
```

There are two more data structures that are made to make type inference easier. The first map is from data constructors and term identifiers to their most general polymorphic types. This map is called `Beta`. The second one is a map from the type constructor names to arities. This map is called Delta. These maps are useful for type inference because the type inference function can quickly look up a type constructor within an expression and acquire its data type and arity.

To construct `Beta`, the semantics uses the already existing T data structure.

### 5.2.1 Transform T into Beta

```
// 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)
```

### 5.2.2 Construct Beta

Beta is a map from the type constructor to its corresponding type.

Example

If the user makes the data type `CusBool` in module *Simp*5, and declares it with the example

```
data CusBool = True2 | False2
```

then the corresponding *tempBeta* should look like this:

```
<tempBeta>
    ModPlusType ( Simp5 , False2 ) |-> forAll ( .Set ,
       CusBool .TyVars )
    ModPlusType ( Simp5 , True2 ) |-> forAll ( .Set ,
       CusBool .TyVars )
</tempBeta>
```

Note how the monomorphic datatype just has an empty $forAll$.

If the user makes the data type CusBool in module Simp5, and declares it with the example

```
data CusBool a b = True2 a | False2 b
```

then the corresponding tempBeta should look like this.

```
<tempBeta>
    ModPlusType ( Simp5 , False2 ) |-> forAll ( b ,
       funtype ( b , CusBool a b ) )
    ModPlusType ( Simp5 , True2 ) |-> forAll ( a ,
       funtype ( a , CusBool a b ) )
</tempBeta>
```

K Code

The following is the K Code.

```
    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:List)) Rest:List),Beta:Map) =>
         makeBeta(TList(ListItem(TObject(ModName,A,B,
             InnerRest)) Rest),Beta[ModPlusType(ModName,Con)
              <- betaParser(E,B,A)])
    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)

    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) => funtype(Tyv, betaParser(Rest, Par, Con))
rule betaParser('optBangATypes('optBangAType('emptyBang
    (.KList),, 'baTypeCon(A:K,, B:K)),, Rest:K), Par:K,
    Con:K) => funtype('baTypeCon(A:K,, B:K), betaParser(
    Rest, Par, Con))
rule betaParser('optBangATypes('optBangAType('emptyBang
    (.KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K,
    Con:K) => funtype(Tyc, betaParser(Rest, Par, Con))
rule betaParser(.OptBangATypes, Par:K, Con:K) => '
    simpleTypeCon(Con,, Par)

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

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

### 5.2.3 Construct Delta

Delta contains the arity of the user defined dataTypes. So if a user defined datatype takes in two parameters, tempDelta will contain the number 2.

Example

If the user makes the data type *CusBool* in module *Simp*5, and declares it with the example

```
data CusBool a b = True2 a | False2 b
```

then the corresponding *tempDelta* should look like this.

```
<tempDelta>
     ModPlusType ( Simp5 , CusBool ) |-> 2
</tempDelta>
```

```
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>
```

## 5.3 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 [13].

## 5.4 Inference Algorithm

The inference function acquires the most general data type for the expression that is fed into it. It inducts down into an expression to the most primitive parts of the expression, and then collects information about each of the primitive parts and how they combine together. Then it passes the information back up the layers and slowly acquires all the information about the most general type of the expression. Much of the inference algorithm is the same as the one introduced in CS 421. This is also used in the definition of Standard ML [14, 15].

```
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 ::= 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>
```

### 5.4.1 Variable Rule

The variable rule is used whenever the inference function encounters a variable. It looks up the variable in the environment, and then acquires the corresponding polymorphic type of the variable in the environment. It then calls `freshInstance` which removes all quantifiers and converts this type to a monomorphic type. This is because we cannot induct any further on the variable `x`, so any quantified variables that the type of `x` contains should instead become free variables. This is because `x` does not act upon anything. The rule then calls unify and sets the fresh type assigned to the variable equal to the output of the `freshInstance` function.

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

```
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>
```

### 5.4.2 Constant Rule

The constant rule is much like the variable rule. The constant rule is used whenever the inference function encounters a constant. It looks up the constant in the `Beta` map, and then acquires the corresponding polymorphic type of the constant in the map. It then calls `freshInstance` which removes all quantifiers and converts this type to a monomorphic type. This is because we cannot induct any further on the constant `c`, so any quantified variables that the type of `c` contains should instead become free variables. This is because `c` does not act upon anything. The rule then calls unify and sets the fresh type assigned to the constant equal to the output of the `freshInstance` function.

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

```
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>
```

### 5.4.3 Lambda Rule

The lambda rule is used for lambda expressions. It overwrites the type of `x` with a fresh type in the environment $\tau_1$, and inducts on e with a different fresh type $\tau_2$. When that corresponds to a substitution, it first looks up the guessed type of the lambda expression in the substitution. It then also looks up the term $\tau_1 \rightarrow \tau_2$ within the substitution as well. This is because if the substitution reveals additional information about these fresh types, we can remove the fresh types and replace them with types that reveal more information about them instead. We then set them equal, because the type of this lambda expression is $\tau_1 \rightarrow \tau_2$.

Also, a lambda expression that takes in multiple variables is instead transformed into multiple lambda expressions that each take in one variable.

Also, functions that take in variables can be desugared to variables that are set equal to lambda terms within a let expression. So, the let-In rule coupled with the lambda rule can infer the type of a function in Haskell, and we do not need an extra rule.

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

```
    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)), genLambda(Apatlist,Ex), guessType(
                TypeIt +Int 1))), .ElemList, Guess, guessType(
                TypeIt),guessType(TypeIt +Int 1)) ...</k>
           <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))))),Sigma)) ...</k>
```

### 5.4.4 Application Rule

The application is similar to the lambda rule. The rule inducts on the left-most expression and places the constraint of the leftmost expression being a function into the recursive call. Then when it gets a substitution, it looks up the fresh type $\tau_1$ within the substitution to get any additional information about the fresh type that we may have. Afterwards it inducts upon the rightmost expression and acquires another substitution. It then composes them and returns that substitution.

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

```
syntax KItem ::= 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))), .ElemList, Gamma, Etwo,
           guessType(TypeIt), .Map) ...</k>
      <typeIterator> TypeIt:Int => TypeIt +Int 1 </
         typeIterator>

rule <k> typeInferenceFunAppli(valValue(mapBagResult(
    Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem,
    guessType(TypeIt:Int), .Map)
       => typeInferenceFunAppli(val(typeInferenceFun(.
           ElemList, gammaSub(Sigmaone, Gamma, .Map),
           Etwo, typeSub(Sigmaone, guessType(TypeIt)))),
           .ElemList, .Map, .K, .K, Sigmaone) ...</k>
```

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

### 5.4.5 IfThenElse Rule

Function guards, cases, and if-then-else are all equivalent. This means that function guards and cases can both be desugared to if-then-else rules and the inference function only has to care about encountering an if-then-else rule to account for all three.

The rule is very similar to the compose rule and the lambda rule. It just keeps inducting into each term and collects information about the types from each of them and then returns back the information it acquired from all three. The only interesting constraint is that the first term must be a Boolean.

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \,|\, \sigma_1 \qquad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \,|\, \sigma_2 \qquad T_2}{\Gamma \vdash \mathrm{if}\, e_1 \,\mathrm{then}\, e_2 \,\mathrm{else}\, e_3 : \tau \,|\, \sigma_3 \circ \sigma_2 \circ \sigma_1} \text{IfThenElse}$$

where $T_2 = \sigma_2 \circ \sigma_1(\Gamma) \vdash e_3 : \sigma_2 \circ \sigma_1(\tau) \,|\, \sigma_3$

```
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), Guess:KItem)
       => typeInferenceFunIfThen(val(typeInferenceFun(.
          ElemList, Gamma, Eone, Bool)), .ElemList,
          Gamma, Etwo, Ethree, Guess, .Map, .Map) ...</k
          >

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

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

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

## 5.5 Mutual Recursion

Note that mutually recursive functions are allowed in Haskell. For example:

$$fx = yx$$

$$yx = fx$$

This set of functions is allowed to compile. When run, the function simply runs forever. This is unlike standard ML [15], which does not allow for mutually recursive functions.

This is also the case for LetIn expressions. These expressions also allow for mutual recursion, where the inside of each declaration within the let expression can refer to other declarations within the let expression.

For instance, an expression that could be written in Haskell is

```
let {f = \x -> y x; y = \x -> f x} in (f 2)
```

In this example, the `f` is an expression that refers to `y` and `y` is an expression that refers to `f`.

### 5.5.1 LetIn Rule

This let rule, unlike the let rule introduced in CS 421, allows for mutual recursion.

The let rule assigns each declaration within the let expression a new fresh monomorphic type. Each of these types is collected in a new environment called $\Psi$. It then inducts into each declaration and keeps collecting type information from each one. As it does this, it keeps composing the substitutions it collects from each declaration and looks up all the type variables within the substitution composition to keep adding the information gathered from each declaration. After it finishes doing this, it finally calls GEN on each type given to the declarations, to make them polymorphic for all the remaining free variables. Afterwards it maps all the variables to these new generated types within the environment. Finally, it inducts within the expression with this new environment, and gives it the type of the substitution of $\tau$.

$$\frac{\sigma_{i+1}(\Psi + \Gamma) \vdash e_i : \Psi(x_i) \mid \sigma_j}{\Gamma \vdash \{\, x_1 = e_1; \cdots\ x_i = e_i; \cdots\ x_n = e_n \,\}\, \mathrm{in\, exp} : \Xi}\ \text{LetIn}$$

where

$$\mathrm{dom}(\Psi) = \{\, x_1, \cdots, x_n \,\}$$

$$\sigma'_0 = \{\}$$

$$\sigma'_i = \sigma_i \circ \sigma'_{j+1}$$

$$\Xi = \bigcup_{i=1}^{n} \{\, x_i \mapsto\ \mathrm{GEN}\,(\sigma'_n(\Gamma), \sigma'_n(\psi_i)) \,\} \mid \sigma'_n$$

$$\frac{\Gamma \vdash \{\, \mathrm{decls} \,\} : \Delta \mid \sigma \qquad \sigma(\Delta + \Gamma) \vdash e : \tau \mid \sigma'}{\Gamma \vdash \mathrm{let}\, \{\, \mathrm{decls} \,\}\, \mathrm{in\, exp} : \tau \mid \sigma' \circ \sigma}\ \text{LetIn}$$

Module

This rule can then be applied to an entire module. An entire module can be inferred in the same way, if the only parts of the module that need to be inferred are the declarations within the module. This is how the semantics can begin the inference algorithm upon a Haskell module, because a module contains everything else.

$$\frac{\Gamma \vdash \{\,\text{decls}\,\} : \Delta \,|\, \sigma}{\Gamma \vdash \text{module}\,\{\,\text{decls}\,\} : \Delta \,|\, \sigma}\,\text{Module}$$

K Code

```
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 -> sigma1(tau2), f3 -> sigma1(tau3),....] +
   sigma1(gamma) |- e2 : sigma1(tau2) | sigma2  [f1 ->
   sigma2 o sigma1(tau1), f2 -> sigma2 o sigma1(tau2),
   f3 -> sigma2 o sigma1(tau3),....] + sigma2 o sigma1(
   gamma) |- e3 : sigma2 o sigma1(tau3) .....  [f1 ->
   gen(sigma_n o sigma2 o sigma1(tau1), sigma_n o
   sigma2 o sigma1(Gamma)), f2 -> gen(tau2), f3 -> gen(
   tau3),....] + gamma |- e : something
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, Beta) ...</k>
     <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
    , TypeIt:Int, OldSigma:Map, Beta:Map)
        => typeInferenceFunLetIn(val(typeInferenceFun(.
            ElemList, Gamma DeclMap, grabLetDeclExp(D,
            Iter), mapLookup(DeclMap, grabLetDeclName(D,
            Iter)))), .ElemList, Gamma, DeclMap, D, E,
            Guess, Iter, TypeIt, OldSigma, Beta) ...</k>
       //=> typeInferenceFunLetIn(val(typeInferenceFun(
           DeclMap, grabLetDeclExp(D, Iter +Int TypeIt),
           Guess)), .ElemList, Gamma, DeclMap, D, E,
           Guess, Iter, TypeIt, OldSigma) ...</k>
       requires Iter <Int (size(DeclMap))

rule <k> typeInferenceFunLetIn(valValue(mapBagResult(
    Sigma:Map)), .ElemList, Gamma:Map, DeclMap:Map, D:
    KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int,
    OldSigma:Map, Beta:Map)
       => typeInferenceFunLetIn(.ElemList, gammaSub(Sigma
           ,Gamma,.Map), gammaSub(Sigma, DeclMap,.Map), D
           , E, typeSub(Sigma, Guess), Iter +Int 1,
           TypeIt, compose(Sigma,OldSigma), Beta) ...</k>
       requires Iter <Int (size(DeclMap))

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

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

## 5.6 Helper Functions

These helper functions are necessary for the inference function.

```
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)
   => makeDeclMap('decls(Rest), TypeIt +Int 1, NewMap[
   Var <- guessType(TypeIt)])
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) =>
   grabLetDeclExp(Righthand, Iter)
     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
```

## 5.7 Fresh Instance

Fresh Instance takes in a polymorphic variable and removes all the quantifiers by taking all quantified variables and replacing them with fresh unused types.

```
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(C, B, Iter)), Iter +Int 1)

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

rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K),
   Iter:Int) => funtype(freshInstanceInner(Repl,A,Iter)
   ,freshInstanceInner(Repl,B,Iter))
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]
```

## 5.8 GEN

The GEN function takes in a monomorphic type and an environment. It will add quantifiers to all free variables in the type that do not appear in the environment.

The rules of GEN are as follows. Note that it just looks for free variables that are in the type that do not appear in the environment and will then add quantifiers.

$$\text{GEN}(\Gamma, \tau) = \forall \alpha_1, \cdots, \alpha_n \,.\, \tau$$

where

$$\{\, \alpha_1, \cdots, \alpha_n \,\} = \text{freevarsty}(\tau) - \text{freevarsenv}(\Gamma)$$

$$\text{freevarsty}('\alpha) = \{\, '\alpha \,\}$$

$$\text{freevarsty}(c) = \{\}$$

where $c$ is a type such as $Int$

$$\text{freevarsty}(c(\tau_1, \cdots, \tau_n)) = \bigcup_{i=1}^{n} \text{freevarsty}(\tau_i)$$

$$\text{freevarsty}(\forall \alpha_1, \cdots, \alpha_n \,.\, \tau) = \text{freevarsty}(\tau) - \{\, \alpha_1, \cdots, \alpha_n \,\}$$

$$\text{freevarsenv}(\Gamma) = \bigcup_{x \in \text{dom}(\Gamma)} \text{freevarsty}(\Gamma(x))$$

The K Code is as follows.

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

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

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, Beta)
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)
```

## 5.9 Unification Algorithm

The goal of the unification algorithm is to provide a substitution. This substitution is used to map variables to monomorphic types. The goal of the substitution is to look up a variable in an expression or an environment

This unification algorithm is provided from CS 421 [14].

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

### 5.9.1 Identity

If the input is empty, then return the identity substitution.

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

### 5.9.2 Non-Identity

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

Delete

If the two terms that are set equal are already equal, then they are not necessary in the substitution map. They can be removed.

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

Decompose

If the two terms use the same constructor and contain the same number of children, this means that each child can be set equal.

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

The substitution should map variables to non-variables, so if the equation has a non-variable mapped to a variable, it should be oriented the other way.

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

Eliminate

This rule removes an equation from the set and adds it to the substitution map if the conditions of the equation are met.

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

Let $\phi = x \mapsto t$

Let $\psi$ = Unif($\phi(S')$)

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

Note:

$$\{\,x \mapsto a\,\} \circ \{\,y \mapsto b\,\} = \{\,y \mapsto (\,\{\,x \mapsto a\,\}(b)\,\} \circ \{\,x \mapsto a\,\}$$

if $y$ not in $a$
[14].

### 5.9.3 K Code

The following is the K code.

```
//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)) ListItem(uniPair(B, D)) Rest:List) //decompose
   rule function type

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 rule
      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(Sigma, A), typeSub(Sigma, B)))

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,
    Tautwo))
rule typeSub(Sigma:Map,Tau:KItem) => Tau [owise]
```

## 5.10 Composition of Substitutions

Composition of substitutions means that if the composition was applied to a type, it would first apply to the rightmost substitution and then afterwards apply to the substitution to the left of it, and so on [13].

The following is a mathematical definition where $\theta$ and $\lambda$ are substitutions.

$$\theta = [t_1/x_1, t_2/x_2, \cdots, t_n/x_n]$$

$$\lambda = [s_1/y_1, s_2/y_2, \cdots, s_m/y_m]$$

$$\lambda \circ \theta = [\lambda(t_1)/x_1, \lambda(t_2)/x_2, \cdots, \lambda(t_n)/x_n]$$

The following is the K Code.

```
    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, Sigmatwo, NewMap, Key, Type)

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

    rule composeIn((Typeone |-> Typetwo:KItem) Sigmaone:Map,
         Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:
        KItem) => composeIn((Typeone |-> Typetwo) Sigmaone,
        Sigmatwo, NewMap[Keyone <- Typetwo], .K, .K)
          requires notBool(Keyone in keys(Sigmaone))

    rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
        Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
        Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]

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

## 5.11 Example

The following is an example inference:

The module called `Simp5` defined a data type called `CusBool` as

```
data CusBool = True2 | False2
```

The data structures for this data type look as follows after running the semantics:

```
<tempBeta>
    ModPlusType ( Simp5 , False2 ) |-> forAll ( .Set ,
        CusBool .TyVars )
    ModPlusType ( Simp5 , True2 ) |-> forAll ( .Set ,
        CusBool .TyVars )
```

```
</tempBeta>
<tempT>
    TList ( ListItem ( TObject ( Simp5 , CusBool , .
       TyVars ,
        ListItem ( InnerTPiece ( True2 , .List , .List ,
             True2
          .OptBangATypes , CusBool ) )
        ListItem ( InnerTPiece ( False2 , .List , .List
           , False2
          .OptBangATypes , CusBool ) ) ) ) )
</tempT>
<tempDelta>
    ModPlusType ( Simp5 , CusBool ) |-> 0
</tempDelta>
```

In the module `Simp5` there is also an example declaration:

```
\y -> (let {f = \x -> x y} in (f (\x -> Simp5.True2)))
```

After running the inference algorithm, the substitution generated for this declaration is as follows:

```
<k>
    mapBagResult (
        guessType ( 0 ) |-> funtype ( guessType ( 1 ) ,
           CusBool .TyVars
          )
        guessType ( 2 ) |-> CusBool .TyVars
        guessType ( 3 ) |-> funtype ( funtype (
           guessType ( 1 ) ,
          guessType ( 5 ) ) , guessType ( 5 ) )
        guessType ( 4 ) |-> funtype ( guessType ( 6 ) ,
           guessType ( 5 )
          )
        guessType ( 6 ) |-> guessType ( 1 )
        guessType ( 7 ) |-> funtype ( guessType ( 9 ) ,
           guessType ( 8 )
          )
        guessType ( 8 ) |-> CusBool .TyVars
        guessType ( 9 ) |-> guessType ( 10 )
        guessType ( 11 ) |-> CusBool .TyVars ) ~>
           inferenceShell (
       .TopDecls )
</k>
```

And the example expression is inferred as

```
    funtype ( guessType ( 1 ) , CusBool .TyVars )
```

which should be the case.

# Chapter 6

# Multiple Module Support

The final part of the semantics presented in this thesis is the beginning of the support of multiple modules. Similar to including files or objects in other programming languages, Haskell modules can include other modules and use functions, types, and type classes declared in the module.

When supporting multiple modules, there are several considerations and additional checks that should be made. This again comes from analyzing the behavior of GHC. Among them:

Modules need to be able 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.

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 directed acyclic graph.

## 6.1 Algorithm

The algorithm is as follows:

1. Construct graph for module inclusion
2. Check graph 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.

`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 multiple files. So instead all the included modules for a program are dumped into one file and are separated by the keyword `<NEXTMODULE>`. This creates a list of modules called `ModuleList`.

The code for graph construction and checking for cycles is included in Appendix D. Building `alpha*` and `beta*`, and desugaring the scope proved difficult and are not included in the semantics presented in this thesis.

In the semantics presented in this thesis, type constructors must be referenced using both the constructor and the module that it was created in. For instance:

```
Simp5.True2
```

# Chapter 7

# Conclusion

The Haskell 2010 report gives a semi-formal outline of the Haskell type system. It also describes it as a Hindley Milner polymorphic type system. However, the original Hindley Milner polymorphic type system is based on a simpler ML system. Haskell, on the other hand, being a modern real world language, is much more complex and requires parsing, preprocessing, modules, expressions, declarations, and other elements. On top of this, Haskell includes mutually recursive functions which need to be inferred. One such inference rule which allows for mutual recursion is presented in this paper. Ultimately, defining a formal syntax and semantics in K allows one to not only automatically convert the formal semantics into proofs in an automatic theorem prover, but also execute example programs using the formal semantics. This allows a formal semantics to be validated by passing the test suite provided for compilers. Defining yet another real word language in K allows the K-framework to become more popular as another community can learn about it and utilize it. The more that real world languages are defined in K, the more potential K has to become a popular language in industry.

# Appendix A

# haskell-syntax.k

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

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

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)]

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


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
```

# Appendix B

# haskell-configuration.k

```
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>
            <typeIterator> 1 </typeIterator>
            <tempAlpha> .K </tempAlpha>
            <tempAlphaMap> .Map </tempAlphaMap>
            <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
```

# Appendix C

# haskell-preprocessing.k

```
//
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
       , Rest)
    rule importListConvert('moduleName(Name:K), 'impDecl(A:K
       ,, Modid:K,, C:K,, D:K)) => ListItem(impObject(Name,
        Modid))
    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
   children
(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,
   stack, graph
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
       ) ListItem(B)
    rule getNodeNeighbors(Node:KItem,ListItem(impObject(A:
       KItem,B:KItem)) Rest:List) => getNodeNeighbors(Node,
        Rest)
         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,B:K)) Rest:List)
*/

//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, graph
    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
        pieces)
    syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type
       constructor, poly type vars, argument sorts, entire
       constr block, type name)

//    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)) => getPreModule(Rest,preModule(AList(ListItem(
   AObject(ModPlusType(ModName,I),C)) M), 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))) => getPreModule(Rest,preModule(AList(M),
   TList(ListItem(TObject(ModName,I,H,makeInnerT(I,H,D)
   )) ListInside))) ...</k>
     <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),C,A)) makeInnerT(A,B,D)

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,Mod))

rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:
   K,B:K,ListItem(InnerTPiece(Ty:K,E:K,F:K,H:K,G:K))
   Inners:List)) Rest:List),J:List,D:Set) =>
                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(TObject(A:K,B:K,C:K)) ~> Rest:K),J:K)
```

```
    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) D, ListItem(A) G)
    rule keyCheck(AList(.List), TList(ListItem(TObject(
       ModName:K, A:K,B:K,C:K)) Rest:List), D:Set, G:List)
       => keyCheck(AList(.List), TList(Rest), SetItem(A) D,
        ListItem(A) G)
    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),
    SetItem(A) D, G)
//    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,T,BigSet)

     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,
         ListItem(B) ListItem(A) D, SetItem(B) SetItem(A) G,
         B, .K,T,S)
     rule aloopCheck(Alpha:Map (H |-> B:KItem), D:List, G:Set
         , H:KItem, .K,T:K,S:Set) => aloopCheck(Alpha,
         ListItem(B) D, SetItem(B) G, B, .K,T,S)

     rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:
         K,S:Set) => aloopCheck(Alpha, .List, .Set, .K,
         lengthCheck(size(G),size(D)),T,G S) //type rename
         loop ERROR
           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 ERROR
           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(A) D)
    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(AList
   (C:List), ListItem(B) D, SetItem(B) G, B)


//    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)

//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)
        ArgsRest:List,E:K,F:K)) InnerRest:List)) TListRest:
       List),AlphaMap:Map,Tset:Set) => argSortCheck(TList(
       ListItem(TObject(ModName,A,B,ListItem(InnerTPiece(C,
       D,ArgsRest,E,F)) InnerRest)) TListRest),AlphaMap,
       Tset)
          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)
        ArgsRest:List,E:K,F:K)) InnerRest:List)) TListRest:
        List),AlphaMap:Map,Tset:Set) => error
          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))
        InnerRest:List)) TListRest:List),AlphaMap:Map,Tset:
        Set) => argSortCheck(TList(ListItem(TObject(ModName,
        A,B,InnerRest)) TListRest),AlphaMap,Tset)

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

    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
        ,Tset:Set) => makeParLists(TList(ListItem(TObject(A,
        PolyList,C)) Rest),ListItem(Arg) Tlist,SetItem(Arg)
        Tset)
    rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,C
        :K)) Rest:List),.K) => parCheck(TList(ListItem(
        TObject(ModName,A:K,B:K,C:K)) Rest:List),
        makeTyVarList(B,.List,.Set))
```

```
rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,
   ListItem(InnerTPiece(C:K,ListItem(Par:KItem) ParRest
   :List,D:K,E:K,F:K)) InnerRest:List)) Rest:List),
   NewSet:Set) =>
     parCheck(TList(ListItem(TObject(ModName,A,B,
        ListItem(InnerTPiece(C,ParRest,D,E,F))
        InnerRest)) Rest),NewSet)
        requires Par in NewSet

rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,
   ListItem(InnerTPiece(C:K,ListItem(Par:KItem) ParRest
   :List,D:K,E:K,F:K)) InnerRest:List)) Rest:List),
   NewSet:Set) => error
        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:List)) Rest:List),NewSet:Set) =>
     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, SetItem(A) NewSet)

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:List)) Rest:List),Beta:Map) =>
```

```
          makeBeta(TList(ListItem(TObject(ModName,A,B,
              InnerRest)) Rest),Beta[ModPlusType(ModName,Con)
               <- betaParser(E,B,A)])
    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:
   List)) Rest:List),Beta:Map) =>
//         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) => funtype(Tyv, betaParser(Rest, Par, Con))
    rule betaParser('optBangATypes('optBangAType('emptyBang
        (.KList),, 'baTypeCon(A:K,, B:K)),, Rest:K), Par:K,
        Con:K) => funtype('baTypeCon(A:K,, B:K), betaParser(
        Rest, Par, Con))
    rule betaParser('optBangATypes('optBangAType('emptyBang
        (.KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K,
        Con:K) => funtype(Tyc, betaParser(Rest, Par, Con))
    rule betaParser(.OptBangATypes, Par:K, Con:K) => '
        simpleTypeCon(Con,, Par)
//    rule betaParser('optBangATypes('optBangAType('
    emptyBang(.KList),, 'atypeGTyCon(Tyc:K)),, Rest:KItem))
    => 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 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, I) ...</k>

    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
```

# Appendix D

# haskell-type-inferencing.k

```
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) => Sigma
    //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>

    //rule typeInferenceFun('aexpGCon(Gcon:K), Alpha:Map, (
        Gcon |-> Sigma:K) Beta:Map, Delta:Map, Gamma:Map,.K
        ,.K) => Sigma //T-App
    //rule typeInferenceFun('aexpGCon(Gcon:K), Alpha:Map,
        Lol:Map, Delta:Map, Gamma:Map,.K,.K) => Sigma //T-
        App
    //     <tempBeta> (Gcon |-> Sigma:K) Beta:Map </tempBeta
        >

    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>

    //lambda rule
//    rule <k> typeInferenceFun(Gamma:Map, 'lambdaFun(
    Apatlist:K,, Ex:K), Guess:KItem)
//          => typeInferenceFun(genGamma(Apatlist,Gamma,
    guessType(TypeIt)), genLambda(Apatlist,Ex), guessType(
    TypeIt +Int 1))
```

```
//          ~> lambdaReturn(Guess,guessType(TypeIt),
   guessType(TypeIt +Int 1)) ...</k>
//         <typeIterator> TypeIt:Int => TypeIt +Int 2 </
   typeIterator>

//    rule <k> Sigma:Map ~> lambdaReturn(Tau:K, Tauone:K,
   Tautwo:K)
//          => compose(uniFun(ListItem(uniPair(typeSub(Sigma
   ,Tau),typeSub(Sigma,funtype(Tauone,Tautwo))))),Sigma)
   ...</k>

   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)), genLambda(Apatlist,Ex), guessType(
             TypeIt +Int 1))), .ElemList, Guess, guessType(
             TypeIt),guessType(TypeIt +Int 1)) ...</k>
         <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))))),Sigma)) ...</k>

    //rule <k> substi(S:Map) ~> lambdaReturn(Tau:K, Tauone:K
       , Tautwo:K)
    //      => S[Tauone] ...</k>


    //syntax KItem ::= appliReturn(Map, K, K, Map)
    //syntax KItem ::= typeChildSub(Map, K) [function]

    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))), .ElemList, Gamma, Etwo,
               guessType(TypeIt), .Map) ...</k>
          <typeIterator> TypeIt:Int => TypeIt +Int 1 </
             typeIterator>

    rule <k> typeInferenceFunAppli(valValue(mapBagResult(
        Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem,
        guessType(TypeIt:Int), .Map)
           => typeInferenceFunAppli(val(typeInferenceFun(.
               ElemList, gammaSub(Sigmaone, Gamma, .Map),
               Etwo, typeSub(Sigmaone, guessType(TypeIt)))),
               .ElemList, .Map, .K, .K, Sigmaone) ...</k>

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

//    rule <k> Sigmaone:Map ~> appliReturn(Gamma:Map, Etwo:
    KItem, guessType(TypeIt:Int), .Map)
//          => typeInferenceFun(gammaSub(Sigmaone, Gamma, .
    Map), Etwo, typeSub(Sigmaone, guessType(TypeIt)))
//          ~> appliReturn(.Map, .K, .K, Sigmaone) ...</k>

//    rule <k> Sigmatwo:Map ~> appliReturn(.Map, .K, .K,
    Sigmaone:Map)
//          => compose(Sigmatwo, Sigmaone) ...</k>
```

```
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), Guess:KItem)
       => typeInferenceFunIfThen(val(typeInferenceFun(.
          ElemList, Gamma, Eone, Bool)), .ElemList,
          Gamma, Etwo, Ethree, Guess, .Map, .Map) ...</k
          >

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

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

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

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 -> sigma1(tau2), f3 -> sigma1(tau3),....] +
   sigma1(gamma) |- e2 : sigma1(tau2) | sigma2  [f1 ->
   sigma2 o sigma1(tau1), f2 -> sigma2 o sigma1(tau2),
   f3 -> sigma2 o sigma1(tau3),....] + sigma2 o sigma1(
   gamma) |- e3 : sigma2 o sigma1(tau3) .....  [f1 ->
   gen(sigma_n o sigma2 o sigma1(tau1), sigma_n o
   sigma2 o sigma1(Gamma)), f2 -> gen(tau2), f3 -> gen(
   tau3),....] + gamma |- e : something
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, Beta) ...</k>
     <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
   , TypeIt:Int, OldSigma:Map, Beta:Map)
       => typeInferenceFunLetIn(val(typeInferenceFun(.
           ElemList, Gamma DeclMap, grabLetDeclExp(D,
           Iter), mapLookup(DeclMap, grabLetDeclName(D,
           Iter)))), .ElemList, Gamma, DeclMap, D, E,
           Guess, Iter, TypeIt, OldSigma, Beta) ...</k>
      //=> typeInferenceFunLetIn(val(typeInferenceFun(
          DeclMap, grabLetDeclExp(D, Iter +Int TypeIt),
          Guess)), .ElemList, Gamma, DeclMap, D, E,
          Guess, Iter, TypeIt, OldSigma) ...</k>
      requires Iter <Int (size(DeclMap))

rule <k> typeInferenceFunLetIn(valValue(mapBagResult(
   Sigma:Map)), .ElemList, Gamma:Map, DeclMap:Map, D:
   KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int,
   OldSigma:Map, Beta:Map)
```

```
      => typeInferenceFunLetIn(.ElemList, gammaSub(Sigma
          ,Gamma,.Map), gammaSub(Sigma, DeclMap,.Map), D
          , E, typeSub(Sigma, Guess), Iter +Int 1,
          TypeIt, compose(Sigma,OldSigma), Beta) ...</k>
      requires Iter <Int (size(DeclMap))

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

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

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(A:K), TypeIt:Int, NewMap:Map)
    => makeDeclMap(A, TypeIt +Int 1, NewMap)
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)
    => makeDeclMap('decls(Rest), TypeIt +Int 1, NewMap[
    Var <- guessType(TypeIt)])
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) =>
   grabLetDeclExp(Righthand, Iter)
     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) ] )
//  => gammaSub(Sigma, Gamma, Newgamma[Key <-
   typeChildSub(Sigma, Type) ] )

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

//rule typeChildSub((guessType(TypeIt) |-> Type:KItem)
   Sigma:Map, guessType(TypeIt:Int)) => Type

//rule typeChildSub(Sigma:Map, guessType(TypeIt:Int)) =>
    guessType(TypeIt)
    //     requires notBool (guessType(TypeIt) in keys(Sigma
       ))

    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(C, B, Iter)), Iter +Int 1)

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

    rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K),
       Iter:Int) => funtype(freshInstanceInner(Repl,A,Iter)
       ,freshInstanceInner(Repl,B,Iter))
    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]
    //syntax KItem ::= setBag(Set)
//    syntax Set ::= listToSet(List, Set) [function]
```

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

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

    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, Beta)
    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)


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

    //Unification

    syntax Map ::= uniFun(List) [function]
    //syntax List ::= uniSub(K,K,K) [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]
```

```
 // syntax KItem ::= Map

 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(SetItem(I:K)) => .K //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(forAll(Svars:List,S:K),
    forAll(.List,T:K))) Rest:List,Sigma:Map) => uniFun(
    ListItem(uniPair(forAll(.List,T),forAll(Svars,S)))
    Rest,Sigma) //orient rule
   //   requires Svars =/=K .List

 //rule uniFun(ListItem(uniPair(guessType(S:Int),forAll(.
    List,T:K))) Rest:List,Sigma:Map) => uniFun(ListItem(
    uniPair(forAll(.List,T:K),guessType(S))) Rest,Sigma)
     //orient rule

// rule uniFun(ListItem(uniPair(forAll(.List,S:K),T:K))
   Rest:List, Sigma:Map) => uniFun(uniSub('aexpQVar(Var)
   ,T,Rest), Sigma['aexpQVar(Var) <- T]) //eliminate
   rule
//      requires notChildVar('aexpQVar(Var:K),T)

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

 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 rule
      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(Sigma, A), typeSub(Sigma, B)))

//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,
    Tautwo))
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, Sigmatwo, NewMap, Key, Type)

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

rule composeIn((Typeone |-> Typetwo:KItem) Sigmaone:Map,
     Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:
    KItem) => composeIn((Typeone |-> Typetwo) Sigmaone,
    Sigmatwo, NewMap[Keyone <- Typetwo], .K, .K)
```

```
          requires notBool(Keyone in keys(Sigmaone))

     rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
        Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
        Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]

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

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

     //rule composeIn((Key:KItem |-> Type:KItem) Sigmaone:Map
        , .Map, NewMap:Map) => composeIn(Sigmaone, .Map,
        NewMap[Key <- mapLookup(Sigmaone, Type)])

     //rule compose(Sigmaone:Map,Sigmatwo:Map) => updateMap(
        Sigmaone,Sigmatwo)
     //rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:
        KItem) Sigmatwo:Map) => compose(Sigmaone[Keytwo <-
        Valtwo][Valtwo <- mapLookup(Sigmaone, Keytwo)],
        Sigmatwo)
     //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
        Sigmatwo:Map, .K) => compose(Sigmaone[Type <-
        mapLookup(Sigmaone, Key)][Key <- Type], Sigmatwo,
        mapLookup(Sigmaone, Key))
     //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
        Sigmatwo:Map) => composeIn(Sigmaone, Sigmatwo,
        mapLookup(Sigmaone, Key))
     //     requires (notBool (Type in values(Sigmaone)))
        andBool (Type =/=K mapLookup(Sigmaone, Key))
     //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
        Sigmatwo:Map) => compose(Sigmaone[Key <- Type][Type
        <- mapLookup(Sigmaone, Key)], Sigmatwo)
     //     requires (notBool (Type in values(Sigmaone)))
        andBool (Type =/=K mapLookup(Sigmaone, Key))
//    rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:
    KItem) Sigmatwo:Map) => compose(Sigmaone[Valtwo <-
    mapLookup(Sigmaone, Keytwo)], Sigmatwo)
//         requires (Valtwo in values(Sigmaone)) andBool (
    Valtwo =/=K mapLookup(Sigmaone, Keytwo))
     //rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:
        KItem) Sigmatwo:Map) => compose((Keytwo |-> Valtwo)
        Sigmaone, Sigmatwo)
     // requires notBool (Keytwo in keys(Sigmaone))
     //rule compose(Sigmaone:Map, .Map) => Sigmaone
     // rule compose(substi(Sone:K,Tone:K),substi(Stwo:K,Ttwo
        :K)) => substi(typeSub(substi(Stwo,Ttwo),Sone),Tone)


    // rule notChildVar('aexpQVar(Var:K),T)


     //T-Var
//     rule typeInferenceFun('funApp(Eone:K,, Etwo:K), Alpha:
    Map, Beta:Map, Delta:Map, Gamma:Map,.K,.K) =>
//         typeInferenceFun('funApp(Eone,, Etwo), Alpha,
    Beta, Delta, Gamma,typeInferenceFun(Eone,Alpha, Beta,
    Delta, Gamma,.K,.K),typeInferenceFun(Etwo,Alpha, Beta,
    Delta, Gamma,.K,.K))
//     rule typeInferenceFun('funApp(Eone:K,, Etwo:K), Alpha:
    Map, Beta:Map, Delta:Map, Gamma:Map, funtype(Tauone:K,
    Tautwo:K), Tauone) => Tautwo

     //T-Lam
//     rule typeInferenceFun('lambdaFun(Apatlist:K,, Ex:K),
    Alpha:Map, Beta:Map, Delta:Map, Gamma:Map,.K,.K) =>
//         typeInferenceFun('lambdaFun(Apatlist,, Ex), Alpha
    , Beta, Delta, Gamma,typeInferenceFun(Ex, Alpha, Beta,
    Delta, genGamma(Apatlist,Gamma),.K,.K),.K)

//     rule typeInferenceFun('lambdaFun(Apatlist:K,, Ex:K),
    Alpha:Map, Beta:Map, Delta:Map, Gamma:Map,Tautwo:K,.K)
    => Tautwo

endmodule
```

# References

[1] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[2] "K-framework." [Online]. Available: http://www.kframework.org/

[3] S. Marlow, "Haskell 2010 report," 2010. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[4] "Ghc website." [Online]. Available: https://www.haskell.org/ghc/

[5] "Free online dictionary of computing." [Online]. Available: http://foldoc.org/type

[6] "Stack overflow." [Online]. Available: https://stackoverflow.com/questions/9345589/guards-vs-if-then-else-vs-cases-in-haskell

[7] B. Pierce, *Types and Programming Languages*, 2002.

[8] R. Hindley, "The principal type scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, Dec. 1969.

[9] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, New York, 1982, pp. 207–212.

[10] "Stanford encyclopedia of philosophy." [Online]. Available: https://plato.stanford.edu/entries/type-theory/

[11] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, June 1940.

[12] J. C. Reynolds, "Towards a Theory of Type Structure," 6 2018. [Online]. Available: https://figshare.com/articles/Towards_a_Theory_of_Type_Structure/6611015

[13] N. Drakos and R. Moore, 2001. [Online]. Available: http://www.mathcs.duq.edu/simon/Fall04/notes-7-4/node3.html

[14] "Cs 421 class notes on inference and unification." [Online]. Available: https://courses.engr.illinois.edu/cs421/fa2018/CS421A/lectures/15-16-poly-type-infer-unif.pdf

[15] R. Milner and M. Tofte, *The Definition of Standard ML*, 1990.