

HASKELL SYNTAX AND STATIC SEMANTICS WRITTEN IN  
K-FRAMEWORK

*Draft of November 27, 2018 at 00:20*

BY

BRADLEY MORRELL

THESIS

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

Urbana, Illinois

Adviser:

Elsa Gunter

# ABSTRACT

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

Subject Keywords: Haskell; Type-System

*Draft of November 27, 2018 at 00:20*

*To my parents, for their love and support.*

# TABLE OF CONTENTS

LIST OF TABLES . . . . .	vi
LIST OF FIGURES . . . . .	vii
LIST OF ABBREVIATIONS . . . . .	viii
CHAPTER 1 INTRODUCTION . . . . .	1
CHAPTER 2 CONTEXT FREE SYNTAX . . . . .	3
2.1 Syntax Explanation . . . . .	4
2.2 Implementation of Section 10.2 . . . . .	5
2.3 Implementation of Section 10.5 . . . . .	8
2.4 Example Test Programs . . . . .	16
CHAPTER 3 CONFIGURATION . . . . .	18
CHAPTER 4 CONTEXT SENSITIVE CHECKS . . . . .	20
4.1 Initial Data Structures . . . . .	22
4.2 Perform Checks . . . . .	25
CHAPTER 5 MULTIPLE MODULE SUPPORT . . . . .	33
5.1 Module Inclusion Tree . . . . .	37
5.2 Leaf DFS . . . . .	39
5.3 Insert AlphaBetaStar . . . . .	41
CHAPTER 6 INFERENCING . . . . .	43
6.1 Data Structures . . . . .	43
6.2 Inferencing Rules . . . . .	46
6.3 Type theory . . . . .	47
6.4 Lambda Calculus . . . . .	47
6.5 Hindley-Milner . . . . .	47
6.6 Definition of Substitution . . . . .	47
6.7 Composition of Substitutions . . . . .	47
6.8 Inferencing Algorithm . . . . .	47
6.9 Unification Algorithm . . . . .	58

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

# LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS



# CHAPTER 1

## INTRODUCTION

One of the inherent problems of engineered systems in general is that the design of the system is not proven to be working. The system could be non-functional at design time. The designer may not fully understand the system environment or may have not considered the behavior of the system in rare circumstances. Then once the design is made, there may be 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 correctness of a system. Without formal methods, the only way to reason about a system is by testing it against different edge cases. Within the context of a programming language, one way a programming language can be formally specified is by defining a syntax and semantics for that language. The operational semantics of a programming language can be thought of as a transition system upon an abstract syntax tree, which is the program itself written in the language, and a state, which is a function from the variables in the tree to the current values of those variables. This way, real and complex programs written in natural looking programming languages can be interpreted as strings written in formal languages. Once a programming language is defined in this way, certain properties and behavior of the language and programs written in the language can be proven. K is a framework for creating the formal specification of a programming language. It then can interpret programs written in the language by running only the rules of the formal operational semantics of the programming language. This allows programs to be 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,

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

## 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 can 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. Some difficulties 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 was 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 didn't parse correctly I could then 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 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 specifies the notation used in the grammar. The notation of 10.1 are always in bold in the grammar. So an example production in the document looks like

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

This means that

```
1 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 Syntax Explanation

There are a lot of parts of the grammar that proposed many challenges to implement 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 my K syntax as split into two options.

```
1 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.

Instead, for each optional sort in the original grammar, I replaced the optional sort with a new sort. For instance, I replaced

```
[ context => ]
```

in the original grammar with a new sort called OptContext. Then I just specified

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

---

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

```
1 "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 okay.

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

```
1 [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.

## 2.2 Implementation of Section 10.2

The following 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.

```
1 // Syntax from haskell 2010 Report
2 // https://www.haskell.org/onlinereport/haskell2010/haskellch10.
  html#x17-17500010
3
4 module HASKELL-SYNTAX
5
6     syntax Integer ::= Token{ ([0-9]+)
7                             | (([0][o]|[0][O])[0-7]+)
8                             | (([0][x] | [0][X])[0-9a-fA-F]+) } [onlyLabel]
9
10    syntax CusFloat ::= Token{ ([0-9]+[\.][0-9]+([e E
11                                ][\+|-]?[0-9]+)?)
12                                | ([0-9]+[e E][\+|-]?[0-9]+) } [
13                                onlyLabel]
14
15    syntax CusChar ::= Token{ [\'] (~[\'\\&]) [\'] } [onlyLabel]
16
17    syntax CusString ::= Token{ [\" ] (~[\"\\&]*) [\" ] } [onlyLabel]
```

---

I ran into issue where a program with a variable called size did not parse. I found out that this is because size is a K keyword. So I just specified that a variable could be a variable token, or size. This reveals an issue in the version of K I was using, where K keywords that appear in example parsed programs won't become tokens.

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

---

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

```

3      ([\! \# \$ \% \& \* \+ \/ \> \? \^][\! \# \$ \% \& \* \+ \.
      \/ \< \= \> \? \@ \\ \^ \|| \- \~ \:]**)
4      | [\-] | [\.]
5      | ([\.] [\! \# \$ \% \& \* \+ \/ \< \= \> \? \@ \\ \^ \|| \- \~
      \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]**)
6      | ([\-][\! \# \$ \% \& \* \+ \. \/ \< \= \? \@ \\ \^ \|| \~
      \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \~
      \:]**)
7      | ([\@][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]+)
8      | ([\~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]+)
9      | ([\\][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]+)
10     | ([\|][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]+)
11     | ([\:] [\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]**)
12     | ([\<][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \~
      \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]**)
13     | ([\=][\! \# \$ \% \& \* \+ \. \/ \< \= \? \@ \\ \^ \|| \~
      \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \|| \-
      \~ \:]**)} [onlyLabel]
14     syntax ConSym ::= Token{[\:] [\! \# \$ \% \& \* \+ \. \/ \<
      \= \> \? \@ \\ \^ \|| \- \~][\! \# \$ \% \& \* \+ \. \/ \<
      \= \> \? \@ \\ \^ \|| \- \~ \:]*} [onlyLabel]

```

---

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

```

1
2     syntax IntFloat ::= "(" Integer ")" [bracket] //NOT
      OFFICIAL SYNTAX

```

```

3          | "(" CusFloat ")" [bracket]

```

---

```

1  syntax Literal ::= IntFloat | CusChar | CusString
2  syntax TyCon  ::= ConId
3  syntax ModId  ::= ConId | ConId "." ModId  [klabel('conModId)
      ]
4  syntax QTyCon ::= TyCon | ModId "." TyCon [klabel('conTyCon)
      ]
5  syntax QVarId ::= VarId | ModId "." VarId [klabel('qVarIdCon
      )]
6  syntax QVarSym ::= VarSym | ModId "." VarSym [klabel('
      qVarSymCon)]
7  syntax QConSym ::= ConSym | ModId "." ConSym [klabel('
      qConSymCon)]
8  /*  syntax QTyCls ::= QTyCon
9  syntax TyCls  ::= ConId
10 /*/
11 syntax TyVars ::= List{TyVar, ""} [klabel('typeVars)] //used
      in SimpleType syntax
12 syntax TyVar  ::= VarId
13 syntax TyVarTuple ::= TyVar "," TyVar      [klabel('
      twoTypeVarTuple)]
14          | TyVar "," TyVarTuple [klabel('
      typeVarTupleCon)]
15
16 syntax Con  ::= ConId | "(" ConSym ")"      [klabel('
      conSymBracket)]
17 syntax Var  ::= VarId | "(" VarSym ")"      [klabel('
      varSymBracket)]
18 syntax QVar ::= QVarId | "(" QConSym ")"    [klabel('
      qVarBracket)]
19 syntax QCon ::= QTyCon | "(" GConSym ")"    [klabel('
      gConBracket)]
20
21 syntax QConOp ::= GConSym | "\"" QTyCon "\"" [klabel('
      qTyConQuote)]
22 syntax QVarOp ::= QVarSym | "\"" QVarId "\"" [klabel('
      qVarIdQuote)]
23 syntax VarOp  ::= VarSym   | "\"" VarId "\"" [klabel('
      varIdQuote)]
24 syntax ConOp  ::= ConSym   | "\"" ConId "\"" [klabel('
      conIdQuote)]
25

```

```
26     syntax GConSym ::= ":" | QConSym
27     syntax Vars ::= Var
28                   | Var "," Vars [klabel('varCon)]
29     syntax VarsType ::= Vars "::" Type [klabel('varAssign)]
30     syntax Ops ::= Op
31                   | Op "," Ops [klabel('opCon)]
32     syntax Fixity ::= "infixl" | "infixr" | "infix"
33     syntax Op ::= VarOp | ConOp
34     syntax CQName ::= Var | Con | QVar
35
36     /* syntax QConId ::= ConId | ModId "." ConId */
37
38     syntax QOp ::= QVarOp | QConOp
```

---

## 2.3 Implementation of Section 10.5

The following introduces the sorts of the context free grammar of the Haskell extended syntax.

### 2.3.1 Modules

We start with modules.

```
1     syntax ModuleName ::= "module" ModId [klabel('moduleName)]
2
3     syntax Module ::= ModuleName "where" Body [klabel('
      module)]
4                   | ModuleName Exports "where" Body [klabel('
      moduleExp)]
5                   | Body [klabel('
      moduleBody)]
6
7     syntax Body ::= "{" ImpDecls ";" TopDecls "}" [klabel('
      bodyimpandtop)]
8                   | "{" ImpDecls "}" [klabel('bodyimpdecls)]
9                   | "{" 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.



### 2.3.2 ImpDecls

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

```
1  syntax ImpDecls ::= List{ImpDecl, ";"} [klabel('impDecls)]
2  syntax ImpDecl  ::= "import" OptQualified ModId OptAsModId
                        OptImpSpec [klabel('impDecl)]
3                        | "" [onlyLabel, klabel('emptyImpDecl)]
4  syntax OptQualified ::= "qualified"
5                        | "" [onlyLabel, klabel('
                        emptyQualified)]
6  syntax OptAsModId  ::= "as" ModId
7                        | "" [onlyLabel, klabel('
                        emptyOptAsModId)]
8
9  syntax OptImpSpec  ::= ImpSpec
10                       | "" [onlyLabel, klabel('
                        emptyOptImpSpec)]
11
12  syntax ImpSpecKey  ::= "(" ImportList OptComma ")"
13  syntax ImpSpec     ::= ImpSpecKey
14                       | "hiding" ImpSpecKey
15
16  syntax ImportList  ::= List{Import, ", " }
17
18  syntax Import      ::= Var
19                       | TyCon CQList
```

---

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

```
1 module Foo where
2 {import Bar
3 }
```

---

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

```
1 import Bar
```

---

Another example program is

```
1 module Simp1 where
2 {import Test
3 }
```

---

The corresponding abstract syntax tree in K is

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

---

Module 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.

### 2.3.3 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.

```
1      syntax TopDecls ::= List{TopDecl, ";"} [klabel('topdeclslist
      )]
2
3      syntax TopDecl ::= Decl [klabel('topdecldecl)]
4          > "type" SimpleType "=" Type [klabel('type)
          ]
5          | "data" OptContext SimpleType OptConstrs
          OptDeriving [klabel('data)]
6          | "newtype" OptContext SimpleType "="
          NewConstr OptDeriving [klabel('newtype)]
7          | "class" OptContext ConId TyVar OptCDecls
          [klabel('class)]
8          | "instance" OptContext QTyCon Inst
          OptIDecls [klabel('instance)]
9          | "default" Types [klabel('default)]
10         | "foreign" FDecl [klabel('foreign)]
```

---

### 2.3.4 Decls

A Decl is any general declaration. So something like

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

---

is a Decl.

```
1  syntax OptDecls ::= "where" Decls | "" [onlyLabel, klabel('
    emptyOptDecls)]
2  syntax Decls ::= "{" DeclsList "}" [klabel('decls)]
3  syntax DeclsList ::= List{Decl, ";"} [klabel('declsList)]
4
5  syntax Decl ::= GenDecl
6                  | FunLhs Rhs [klabel('declFunLhsRhs)]
7                  | Pat Rhs [klabel('declPatRhs)]
8
9  syntax GenDecl ::= VarsType
10                    | Vars "::" Context "=>" Type [klabel('
    genAssignContext)]
11                    | Fixity Ops
12                    | Fixity Integer Ops
13                    | "" [onlyLabel, klabel('emptyGenDecl)]
14
15  syntax FunLhs ::= Var APatList [klabel('varAPatList)]
16                  | Pat VarOp Pat [klabel('patVarOpPat)]
17                  | "(" FunLhs ")" APatList [klabel('
    funlhsAPatList)]
18
19  syntax Rhs ::= "=" Exp OptDecls [klabel('eqExpOptDecls)]
20              | GdRhs OptDecls [klabel('gdRhsOptDecls)]
21
22  syntax GdRhs ::= Guards "=" Exp
23                  | Guards "=" Exp GdRhs
24
25  syntax Guards ::= "|" GuardList
26
27  syntax GuardList ::= Guard | Guard "," GuardList [klabel('
    guardListCon)]
28
29  syntax Guard ::= Pat "<-" InfixExp
30                  | "let" Decls
31                  | InfixExp
32
33  //definition of exp
34  syntax Exp ::= InfixExp
35              > InfixExp "::" Type [klabel('expAssign)]
```

```
25         | InfixExp "::" Context "=>" Type  [klabel('
           expAssignContext)]
26
27     syntax InfixExp ::= LExp
28         > "-" InfixExp  [klabel('minusInfix)]
29         > LExp QOp InfixExp
```

---

### 2.3.5 LExp

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.

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

---

### 2.3.6 AExp

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

```
1
2     syntax OptSemicolon ::= ";" | "" [onlyLabel, klabel('
           emptySemicolon)]
3     syntax OptComma ::= "," | "" [onlyLabel, klabel('
           emptyComma)]
4
5     syntax AExp ::= QVar [klabel('aexpQVar)]
6         | GCon [klabel('aexpGCon)]
7         | Literal [klabel('aexpLiteral)]
8         > AExp AExp [left, klabel('funApp)]
9         > QCon "{" FBindList "}"
10        | AExp "{" FBindList "}" //aexp cannot be qcon
           UNFINISHED
```

```
11          //Liyi: first, does not understand the
              syntax, it is the Qcon {FBindlist}
12          //or QCon? Second, place a check in
              preprosssing.
13          //and also check the Fbindlist here
              must be at least one argument
14          > "(" Exp ")" [bracket]
15          | "(" ExpTuple ")"
16          | "[" ExpList "]"
17          | "[" Exp OptExpComma ".." OptExp "]"
18          | "[" Exp "|" Quals "]"
19          | "(" InfixExp QOp ")"
20          | "(" QOp InfixExp ")" //qop cannot be - (
              minus) UNFINISHED
21          //Liyi: place a check here to check
              if QOp is a minus
```

---

```
1
2  syntax OptExpComma ::= "," Exp | "" [onlyLabel, klabel('
    emptyExpComma)]
3  syntax OptExp ::= Exp | "" [onlyLabel, klabel('emptyExp)]
4
5  syntax ExpList ::= Exp | Exp "," ExpList [right]
6  syntax ExpTuple ::= Exp "," Exp [right, klabel('
    twoExpTuple)]
7                      | Exp "," ExpTuple [right, klabel('
    expTupleCon)]
8
9  //constr datatypes
10 syntax OptConstrs ::= "=" Constrs [klabel('nonemptyConstrs)
    ] | "" [onlyLabel, klabel('emptyConstrs)]
11 syntax Constrs ::= Constr [klabel('singleConstr)] |
    Constr "|" Constrs [klabel('multConstr)]
12 syntax Constr ::= Con OptBangATypes [klabel('constrCon)
    ] // (arity con = k, k 0) UNFINISHED
13                      | SubConstr ConOp SubConstr
14                      | Con "{" FieldDeclList "}"
15
16 syntax NewConstr ::= Con AType [klabel('newConstrCon)]
17                      | Con "{" Var "::" Type "}"
18
19 syntax SubConstr ::= BType | "!" AType
20 syntax FieldDeclList ::= List{FieldDecl, ","}
```

```
21     syntax FieldDecl ::= VarsType
22                           | Vars "::" "!" AType
23
24
25     syntax OptBangATypes ::= List{OptBangAType, " "} [klabel('
      optBangATypes)]
26     syntax OptBangAType ::= OptBang AType [klabel('optBangAType)
      ]
27     syntax OptBang ::= "!" | "" [onlyLabel, klabel('emptyBang)]
28
29     syntax OptContext ::= Context "=>" | "" [onlyLabel, klabel('
      emptyContext)]
30     syntax Context ::= Class
31                           | "(" Classes ")"
32
33     syntax Classes ::= List{Class, ", "}
34
35     syntax SimpleClass ::= QTyCon TyVar [klabel('classCon)]
36
37     syntax Class          ::= SimpleClass
38                           | QTyCon "(" TyVar ATypeList ")"
39                           //Liyi: a check in preprocessing to
                           //      check if the Atype list is
                           //      empty
40                           //it must have at least one item
41
42     //define type and simple type
43     syntax SimpleType ::= TyCon TyVars [klabel('simpleTypeCon)
      ]
44     syntax Type ::= BType
45                   | BType "->" Type [klabel('typeArrow)]
46     syntax BType ::= AType
47                   | BType AType [klabel('baTypeCon)]
48
49     syntax ATypeList ::= List{AType, ""} [klabel('atypeList)]
50
51     syntax AType ::= GTyCon [klabel('
      atypeGTyCon)]
52                   | TyVar [klabel('
      atypeTyVar)]
53                   | "(" TypeTuple ")" [klabel('
      atypeTuple)]
```

```

54             | "[" Type "]"                [klabel('tyList)
              ]
55             | "(" Type ")"                [bracket]
56 syntax TypeTuple ::= Type "," Type       [right,klabel('
              twoTypeTuple)]
57             | Type "," TypeTuple         [klabel('
              typeTupleCon)]
58 syntax Types ::= List{Type, ","}
59
60 syntax GConCommas ::= "," | "," GConCommas
61 syntax GConCommon ::= "()" | "[]" | "(" GConCommas ")" //was
              incorrect syntax
62 syntax GTyCon ::= QTyCon
63             | GConCommon
64             | "(->)"
65
66 syntax GCon ::= GConCommon
67             | QCon
68
69 //inst definition
70 syntax Inst ::= GTyCon
71             | "(" GTyCon TyVars ")" //TyVars must be
              distinct UNFINISHED
72             | "(" TyVarTuple ")" //TyVars must be
              distinct
73             | "[" TyVar "]" [klabel('tyVarList)]
74             | "(" TyVar "->" TyVar ")" //TyVars must be
              distinct
75 //pat definition
76 syntax Pat ::= LPat QConOp Pat
77             | LPat
78
79 syntax LPat ::= APat
80             | "-" IntFloat [klabel('minusPat)]
81             | GCon APatList [klabel('lpatCon)]//arity
              gcon = k UNFINISHED
82
83 syntax APatList ::= APat | APat APatList [klabel('apatCon)]
84
85 syntax APat ::= Var [klabel('apatVar)]
86             | Var "@" APat
87             | GCon
88             | QCon "{" FPats "}"

```

```
89             | Literal [klabel('apatLiteral)]
90             | "_"
91             | "(" Pat ")" [bracket]
92             | "(" PatTuple ")"
93             | "[" PatList "]"
94             | "~" APat
95
96 syntax PatTuple ::= Pat "," Pat [klabel('twoPatTuple
    )]
97             | Pat "," PatTuple [klabel('patTupleCon
    )]
98 syntax PatList ::= Pat
99             | Pat "," PatList [klabel('patListCon)
    ]
100
101 syntax FPats ::= List{FPat, ","}
102 syntax FPat ::= QVar "=" Pat
103
104 //definition of quals
105 syntax Quals ::= Qual | Qual "," Quals [klabel('qualCon)]
106
107 syntax Qual ::= Pat "<-" Exp
108             | "let" Decls
109             | Exp
110
111 //definition of alts
112 syntax Alts ::= Alt | Alt ";" Alts
113
114 syntax Alt ::= Pat "->" Exp [klabel('altArrow)]
115             | Pat "->" Exp "where" Decls
116             | "" [onlyLabel, klabel('emptyAlt)]
117
118 //definition of stmts
119 syntax Stmts ::= StmtList Exp OptSemicolon
120 syntax StmtList ::= List{Stmt, ""}
121 syntax Stmt ::= Exp ";"
122             | Pat "<-" Exp ";"
123             | "let" Decls ";"
124             | ";"
125
126 //definition of fbind
127 syntax FBindList ::= List{FBind, ","}
128 syntax FBind ::= QVar "=" Exp
```

---



## 2.4 Example Test Programs

An example program is

```
1 module Simp1 where
2   {import Test
3   }
```

The corresponding abstract syntax tree in K is

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

Module 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.

## CHAPTER 3

# CONFIGURATION

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 all the code and 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.

The following is the configuration of my Haskell semantics.

```
1 requires "haskell-syntax.k"
2
3 module HASKELL-CONFIGURATION
4     imports HASKELL-SYNTAX
5
6     syntax KItem ::= "startImportRecursion"
7     syntax KItem ::= callInit(K)
8     //syntax KItem ::= initPreModule(K) [function]
9     //syntax KItem ::= tChecker(K) [function]
10
11     configuration
12         <T>
13             <k> $PGM:ModuleList ~> startImportRecursion </k>
14             <tempModule> .K </tempModule>
15             <tempCode> .K </tempCode>
```

---

The  $< k >$  cell is the cell that computation takes place in. The program initially tempModule is the name of the current module.

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

---

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

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

---

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

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

---

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

```
1      <tempBeta> .Map </tempBeta>
2      <tempT> .K </tempT>
3      <tempDelta> .Map </tempDelta>
4      <tempAlphaStar> .K </tempAlphaStar>
5      <tempBetaStar> .K </tempBetaStar>
6      <importTree> .List </importTree>
7      <recurImportTree> .List </recurImportTree>
8      <impTreeVMap> .Map </impTreeVMap>
9      <modules> //static information about a module
10         <module multiplicity="*">
11             <moduleName> .K </moduleName>
12             <moduleAlphaStar> .K </moduleAlphaStar>
13             <moduleBetaStar> .K </moduleBetaStar>
14             <moduleImpAlphas> .List </moduleImpAlphas>
15             <moduleImpBetas> .List </moduleImpBetas>
16             <moduleCompCode> .K </moduleCompCode>
17             <moduleTempCode> .K </moduleTempCode>
18             <imports> .Set </imports>
19             <classes> //static information about a
                module
20                 <class multiplicity="*">
21                     <className> .K </className>
22                 </class>
23             </classes>
24         </module>
25     </modules>
26 </T>
27
28 endmodule
```

---

# CHAPTER 4

## CONTEXT SENSITIVE CHECKS

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

```
1 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.

```
1 type Username = String
```

---

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

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

---

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

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

---

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

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

---

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

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

---

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

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

The second data structure I made is called T. T holds the user defined types created using data and newtype.

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

---

T is a list of TObjects, each TObjct represents a single user defined datatype. It holds the name, the list of polymorphic parameters, and a list of inner T pieces. An inner T piece represents an option of what a type could be. It

consists of a type constructor, a list of polymorphic parameters required for this option, the fields for this option, the entire subtree of the AST for this option unedited, and the type name again. I then used these data structures to perform these checks, and afterwards will transform them into a new data structure to perform type inferencing.

## 4.1 Initial Data Structures

The following parses the tree and searches for the

```
1      //get alpha and beta
2      syntax KItem ::= Module(K, K)
3      syntax KItem ::= preModule(K,K) //(alpha, T)
4
5      // STEP 1 CONSTRUCT T AND ALPHA
6      // alpha = type
7      // T = newtype and data, temporary data structure
8
9      syntax KItem ::= initPreModule(K) [function]
10     syntax KItem ::= getPreModule(K, K) [function] //(Current
        term, premodule)
11     syntax KItem ::= makeT (K,K,K,K)
12
13     syntax KItem ::= fetchTypes (K,K,K,K)
14
15     syntax List ::= makeInnerT (K,K,K) [function] //LIST
16     syntax List ::= getTypeVars(K) [function] //LIST
17
18     syntax KItem ::= getCon(K) [function]
19     syntax List ::= getArgSorts(K) [function] //LIST
20
21     syntax KItem ::= AList(K)
22     syntax KItem ::= AObject(K,K) //(1st -> 2nd) map without
        idempotency
23     syntax KItem ::= ModPlusType(K,K)
24
25     syntax KItem ::= TList(K) //list of T objects for every new
        type introduced by data and newtype
26     syntax KItem ::= TObject(K,K,K,K) //(module name, type name,
        entire list of poly type vars, list of inner T pieces)
```

```
27     syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type constructor,
        poly type vars, argument sorts, entire constr block,
        type name)
28
29 //     rule initPreModule('module(I:ModuleName,, J:K)) =>
        getPreModule(J,preModule(AList(.List),TList(.List)))
30 //     rule initPreModule('moduleExp(I:ModuleName,, L:K,, J:K))
        => getPreModule(J,preModule(AList(.List),TList(.List)))
31 //     rule initPreModule('moduleBody(J:Body)) => getPreModule(J,
        preModule(AList(.List),TList(.List)))
32
33     rule initPreModule(J:K) => getPreModule(J,preModule(AList(.
        List),TList(.List)))
34
35     rule getPreModule('bodytopdecls(I:K), J:K) => getPreModule(I
        ,J)
36     rule getPreModule('topdeclslist('type(A:K,, B:K,, Rest:K),J
        :K) => fetchTypes(A,B,Rest,J) //constructalpha
37
38
39     rule getPreModule('topdeclslist('data(A:K,, B:K,, C:K,, D:K)
        ,, Rest:K),J:K) => makeT(B,C,Rest,J)
40     rule getPreModule('topdeclslist('newtype(A:K,, B:K,, C:K,, D
        :K,, Rest:K),J:K) => makeT(B,C,Rest,J)
41
42
43     rule getPreModule('topdeclslist('topdecldecl(A:K,, Rest:K),
        J:K) => getPreModule(Rest,J)
44     rule getPreModule('topdeclslist('class(A:K,, B:K,, C:K,, D:K
        ),, Rest:K),J:K) => getPreModule(Rest,J)
45     rule getPreModule('topdeclslist('instance(A:K,, B:K,, C:K,,
        D:K),, Rest:K),J:K) => getPreModule(Rest,J)
46     rule getPreModule('topdeclslist('default(A:K,, B:K,, C:K,, D
        :K),, Rest:K),J:K) => getPreModule(Rest,J)
47     rule getPreModule('topdeclslist('foreign(A:K,, B:K,, C:K,, D
        :K),, Rest:K),J:K) => getPreModule(Rest,J)
48     rule getPreModule(.TopDecls,J:K) => J
49
50 //rule getPreModule('module(I:ModuleName,L:K, J:K)) =>
        preModule(J)
51
52     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>
53     <tempModule> ModName:KItem </tempModule>
54
55 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>
56     <tempModule> ModName:KItem </tempModule>
57
58 rule makeInnerT(A:K,B:K,'nonemptyConstrs(C:K)) => makeInnerT
      (A,B,C)
59 rule makeInnerT(A:K,B:K,'singleConstr(C:K)) => ListItem(
      InnerTPiece(getCon(C),getTypeVars(C),getArgSorts(C),C,A))
60 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)
61
62 rule getTypeVars('constrCon(A:K,, B:K)) => getTypeVars(B)
63 rule getTypeVars('optBangATypes(A:K,, Rest:K)) =>
      getTypeVars(A) getTypeVars(Rest)
64 rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K))
      => getTypeVars(Rest)
65 rule getTypeVars('atypeGTyCon(A:K)) => .List
66 rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
67 rule getTypeVars(.OptBangATypes) => .List
68
69 //rule getCon('emptyConstrs()) => .K
70 //rule getCon('nonemptyConstrs(A:K)) => getCon(A)
71 rule getCon('constrCon(A:K,, B:K)) => A
72
73 //rule getArgSorts('constrCon(A:K,, B:K)) => B
74 rule getArgSorts('constrCon(A:K,, B:K)) => getArgSorts(B)
75 rule getArgSorts('optBangATypes(A:K,, Rest:K)) =>
      getArgSorts(A) getArgSorts(Rest)
76 rule getArgSorts('optBangAType('emptyBang(.KList),, Rest:K))
      => getArgSorts(Rest)
77 rule getArgSorts('atypeGTyCon(A:K)) => ListItem(A)
78 rule getArgSorts('atypeTyVar(A:K)) => .List
79 rule getArgSorts(.OptBangATypes) => .List
80
```



```
81 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
82
83 rule <k> preModule(A:K,T:K) => startTTransform ...</k>
84   <tempAlpha> OldAlpha:K => A </tempAlpha>
85   <tempT> OldT:K => T </tempT>
```

---

## 4.2 Perform Checks

```
1 // STEP 2 PERFORM CHECKS
2
3 syntax KItem ::= "error"
4
5 syntax KItem ::= "startChecks"
6 syntax KItem ::= "checkNoSameKey"
7   //Keys of alpha and keys of T should be unique
8 syntax KItem ::= "checkTypeConsDontCollide"
9   //Make sure typeconstructors do not collide in T
10 syntax KItem ::= "makeAlphaMap"
11   //make map for alpha
12 syntax KItem ::= "checkAlphaNoLoops"
13   //alpha check for no loops
14   //check alpha to make sure that everything points to a T
15 syntax KItem ::= "checkArgSortsAreTargets"
16   //Make sure argument sorts [U] [W,V] are in the set
17   //of keys of alpha and targets of T, (keys of T)
17 syntax KItem ::= "checkParUsed"
18 //NEED TO CHECK all the polymorphic parameters from right appear
19 //on left. RIGHT SIDE ONLY
19 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
20
21 // rule <k> preModule(A:K,T:K) => startChecks ...</k>
22 //   <tempAlpha> OldAlpha:K => A </tempAlpha>
23 //   <tempT> OldT:K => T </tempT>
24
25
26 /* rule <k> performNextChecks
27   => checkUseVars
28   ~> (checkLabelUses
29   ~> (checkBlockAddress(.K)
```

```

30      ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS
        -Set TES)
31      ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
32      ~> (checkAllExpInFromInvoke(.K, TES)
33      ~> (checkLandingpad
34      ~> checkLandingDomResumes)))))) ...</k> */
35
36  rule <k> startChecks
37      => checkNoSameKey
38      ~> (checkTypeConsDontCollide
39      ~> (makeAlphaMap
40      ~> (checkAlphaNoLoops
41      ~> (checkArgSortsAreTargets
42      ~> (checkParUsed)))))) ...</k>

```

---

## 4.2.1 Collision Check

```

1
2  rule <k> checkTypeConsDontCollide
3      => tyConCollCheck(T,.List,.Set) ...</k>
4      <tempT> T:K </tempT>
5
6  //syntax KItem ::= tChecker(K) [function]
7  syntax KItem ::= tyConCollCheck(K,K,K) [function] //(TList,
      List of Tycons,Set of Tycons)
8  syntax KItem ::= lengthCheck(K,K) [function]
9  //syntax KItem ::= tyConCollCheck(K,K,K) [function]
10 //syntax K ::= innerCollCheck(K) [function]
11 //syntax K ::= tyConCollCheckPasser(K, K) [function]
12
13 //rule tChecker(preModule(Alpha:Map,T:K,Mod:K)) =>
      tyConCollCheck(innerCollCheck(T),preModule(Alpha,T,Mod))
14
15 //rule tyConCollCheck(.K,preModule(Alpha:Map,H:K,Mod:K)) =>
      tyConCollCheck(innerCollCheck(H),preModule(Alpha,H,Mod))
16
17 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) =>
18     tyConCollCheck(TList(ListItem(TObject(
      ModName,A,B,Inners)) Rest),ListItem(Ty) J
      , SetItem(Ty) D)

```

```
19     rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:
      K,.List)) Rest:List),J:List,D:Set) =>
20         tyConCollCheck(TList(Rest),J,D)
21     rule tyConCollCheck(TList(.List),J:List,D:Set) =>
22         lengthCheck(size(J),size(D))
23
24     rule lengthCheck(A:Int, B:Int) => .K
25         requires A ==Int B
26
27     rule lengthCheck(A:Int, B:Int) => error
28         requires A /=Int B
29
30     //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)
```

---

## 4.2.2 Key Check

```
1     syntax KItem ::= keyCheck(K,K,K,K) [function] //(Alpha, T,
      List of names, Set of names)
2
3     rule <k> checkNoSameKey
4         => keyCheck(A, T, .Set, .List) ...</k>
5         <tempAlpha> A:K </tempAlpha>
6         <tempT> T:K </tempT>
7     //rule <k> checkAlphaNoSameKey
8     //         => akeyCheck(.K, .Set) ...</k>
9
10    rule keyCheck(Alis(ListItem(AObject(A:K,B:K)) C:List), T:K,
      D:Set, G:List) => keyCheck(Alis(C), T, SetItem(A) D,
      ListItem(A) G)
11    rule keyCheck(Alis(.List), TList(ListItem(TObject(ModName:K
      , A:K,B:K,C:K)) Rest:List), D:Set, G:List) => keyCheck(
      Alis(.List), TList(Rest), SetItem(A) D, ListItem(A) G)
12    rule keyCheck(Alis(.List), TList(.List), D:Set, G:List) =>
      lengthCheck(size(G),size(D))
13
14
15    syntax KItem ::= makeAlphaM(K,K) [function] //(Alpha,
      AlphaMap)
16    syntax KItem ::= tAlphaMap(K) //(AlphaMap) temp alphamap
17
```

```
18     rule <k> makeAlphaMap
19         => makeAlphaM(A, .Map) ...</k>
20     <tempAlpha> A:K </tempAlpha>
21
22     rule makeAlphaM(AList(ListItem(AObject(A:K,B:K)) C:List), M:
23         Map) => makeAlphaM(AList(C), M[A <- B])
24
25     rule makeAlphaM(AList(.List), M:Map) => tAlphaMap(M)
26
27     rule <k> tAlphaMap(M:K) => .K ...</k>
28     <tempAlphaMap> OldAlphaMap:K => M </tempAlphaMap>
29
30 //     syntax KItem ::= tkeyCheck(K,K,K) [function] //(T,List of
31 //         T,Set of T)
32
33 //     rule <k> checkTNoSameKey
34 //         => tkeyCheck(T, .Set, T) ...</k>
35 //     <tempT> T:K </tempT>
36
37 //     rule tkeyCheck(TList(ListItem(TObject(A:K,B:K,C:K)) Rest:
38 //         List), D:Set, G:K) => tkeyCheck(TList(Rest), SetItem(A) D, G)
39 //     rule tkeyCheck(TList(.List), D:Set, TList(G:List)) =>
40         lengthCheck(size(G),size(D))
```

---

### 4.2.3 Alpha Loop Check

```
1     syntax KItem ::= aloopCheck(K,K,K,K,K,K,K) [function] //(
2         Alpha,List of Alpha,Set of Alpha,CurrNode,lengthcheck,T,
3         BigSet)
4
5     rule <k> checkAlphaNoLoops
6         => aloopCheck(A,.List,.Set,.K,.K,T,.Set) ...</k>
7     <tempAlphaMap> A:K </tempAlphaMap>
8     <tempT> T:K </tempT>
9
10 //alooCheck set and list to check cycles
11
12     rule aloopCheck(Alpha:Map (A:KItem |-> B:KItem), D:List, G:
13         Set, .K, .K,T:K,S:Set) => aloopCheck(Alpha, ListItem(B)
14         ListItem(A) D, SetItem(B) SetItem(A) G, B, .K,T,S)
15
16     rule aloopCheck(Alpha:Map (H |-> B:KItem), D:List, G:Set, H:
17         KItem, .K,T:K,S:Set) => aloopCheck(Alpha, ListItem(B) D,
18         SetItem(B) G, B, .K,T,S)
```

```
12     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
13     requires (notBool H in keys(Alpha)) andBool (H in
      typeSet(T, .Set) orBool H in S)
14
15     rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:K,S:
      Set) => error //terminal alpha rename is not in T ERROR
16     requires (notBool H in keys(Alpha)) andBool (notBool (H
      in typeSet(T, .Set) orBool H in S))
17
18
19     syntax Set ::= typeSet(K,K) [function] //(K, KSet)
20     rule typeSet(TList(ListItem(TObject(ModName:K, A:K,B:K,C:K))
      Rest:List), D:Set) => typeSet(TList(Rest), SetItem(A) D)
21     rule typeSet(TList(.List), D:Set) => D
22
23 //     rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K) =>
      keys(Alpha) ~> H
24 //         requires notBool H in keys(Alpha)
25
26     rule aloopCheck(.Map, .List, .Set, .K, .K,T:K, S:Set) => .K
27 //     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)
28
29
30 //     syntax KItem ::= TList(K) //list of T objects for every
      new type introduced by data and newtype
31 //     syntax KItem ::= TObject(K,K,K) //(type name, entire list
      of poly type vars, list of inner T pieces)
32 //     syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type
      constructor, poly type vars, argument sorts, entire constr
      block, type name)
```

---

#### 4.2.4 Arg Sort Check

```
1 //Make sure argument sorts [U] [W,V] are in the set of keys of
      alpha and targets of T, (keys of T)
2
3     syntax KItem ::= argSortCheck(K,K,K) [function] //(T,
      AlphaMap)
```

```

4
5     rule <k> checkArgSortsAreTargets
6         => argSortCheck(T,A,typeSet(T,.Set)) ...</k>
7         <tempAlphaMap> A:K </tempAlphaMap>
8         <tempT> T:K </tempT>
9
10    rule argSortCheck(TList(ListItem(TObject(ModName:K, A:K,B:K,
11        ListItem(InnerTPiece(C:K,D:K,ListItem(Arg:KItem) ArgsRest
12        :List,E:K,F:K)) InnerRest:List)) TListRest:List),AlphaMap
13        :Map,Tset:Set) => argSortCheck(TList(ListItem(TObject(
14        ModName,A,B,ListItem(InnerTPiece(C,D,ArgsRest,E,F))
15        InnerRest)) TListRest),AlphaMap,Tset)
16        requires ((Arg in keys(AlphaMap)) orBool (Arg in Tset))
17
18    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,
19        ListItem(InnerTPiece(C:K,D:K,ListItem(Arg:KItem) ArgsRest
20        :List,E:K,F:K)) InnerRest:List)) TListRest:List),AlphaMap
        :Map,Tset:Set) => error
        requires (notBool ((Arg in keys(AlphaMap)) orBool (Arg
            in Tset)))
21
22    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,
23        ListItem(InnerTPiece(C:K,D:K,.List,E:K,F:K)) InnerRest:
24        List)) TListRest:List),AlphaMap:Map,Tset:Set) =>
25        argSortCheck(TList(ListItem(TObject(ModName,A,B,InnerRest
26        )) TListRest),AlphaMap,Tset)
27
28    rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,.
29        List)) TListRest:List),AlphaMap:Map,Tset:Set) =>
30        argSortCheck(TList(TListRest),AlphaMap,Tset)
31
32    rule argSortCheck(TList(.List),AlphaMap:Map,Tset:Set) => .K

```

---

#### 4.2.5 Par Check

```

1 //NEED TO CHECK all the polymorphic parameters from right appear
  on left. RIGHT SIDE ONLY
2 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
3
4 syntax KItem ::= parCheck(K,K) [function] //(T,AlphaMap)
5 syntax KItem ::= makeTyVarList(K,K,K) [function] //(TyVars,
  NewList)

```

```
6      syntax KItem ::= lengthRet (K,K,K) [function]
7
8      rule <k> checkParUsed
9          => parCheck(T,.K) ...</k>
10         <tempT> T:K </tempT>
11
12     //rule makeParLists (TList (ListItem (TObject (A:K, ListItem (Arg:
13         KItem) PolyList:List, C:K)) Rest:List), Tlist:List, Tset:Set
14         ) => makeParLists (TList (ListItem (TObject (A, PolyList, C))
15         Rest), ListItem (Arg) Tlist, SetItem (Arg) Tset)
16
17     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K, C:K))
18         Rest:List), .K) => parCheck (TList (ListItem (TObject (
19         ModName, A:K, B:K, C:K)) Rest:List), makeTyVarList (B, .List, .
20         Set))
21
22     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K,
23         ListItem (InnerTPiece (C:K, ListItem (Par:KItem) ParRest:List
24         , D:K, E:K, F:K)) InnerRest:List)) Rest:List), NewSet:Set) =>
25         parCheck (TList (ListItem (TObject (ModName, A, B, ListItem (
26         InnerTPiece (C, ParRest, D, E, F)) InnerRest)) Rest),
27         NewSet)
28         requires Par in NewSet
29
30     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K,
31         ListItem (InnerTPiece (C:K, ListItem (Par:KItem) ParRest:List
32         , D:K, E:K, F:K)) InnerRest:List)) Rest:List), NewSet:Set) =>
33         error
34         requires notBool (Par in NewSet)
35
36     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K,
37         ListItem (InnerTPiece (C:K, .List, D:K, E:K, F:K)) InnerRest:
38         List)) Rest:List), NewSet:Set) =>
39         parCheck (TList (ListItem (TObject (ModName, A, B, InnerRest))
40         Rest), NewSet)
41
42     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K, .List
43         )) Rest:List), NewSet:Set) =>
44         parCheck (TList (Rest), NewSet)
45
46     rule parCheck (TList (.List), NewSet:Set) => .K
47
48     rule makeTyVarList ('typeVars (A:K, , Rest:K), NewList:List,
49         NewSet:Set) => makeTyVarList (Rest, ListItem (A) NewList,
```

```

    SetItem(A) NewSet)
31
32   rule makeTyVarList(.TyVars, NewList:List, NewSet:Set) =>
    lengthRet(size(NewList), size(NewSet), NewSet)
33
34   rule lengthRet(A:Int, B:Int, C:K) => C
35       requires A ==Int B
36
37   rule lengthRet(A:Int, B:Int, C:K) => error
38       requires A !=Int B
39
40   //rule argSortCheck(TList(ListItem(TObject(A:K, B:K, C:K))


---


```



# CHAPTER 5

## MULTIPLE MODULE SUPPORT

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

```
1 File 1:
2 module File1 where
3 {data A = B
4 }
5 File 2:
6 module File2 where
7 {data A = B
8 ;data C = B A
9 }
```

---

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

```
1 File 1:
2 module File1 where
3 {data A = B
4 }
5 File 2:
6 module File2 where
7 {data A = B
```

```
8 }
9 File 3:
10 module File3 where
11 {import File1
12 ;import File2
13 ;data C = B A
14 }
```

---

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

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

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

```
1 // CUSTOM SYNTAX NOT PART OF OFFICAL HASKELL
2
3     syntax ModuleList ::= Module [klabel('modListSingle)] |
        Module "<NEXTMODULE>" ModuleList [klabel('modList)]
```

---

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

```
1 //
2 requires "haskell-syntax.k"
3 requires "haskell-configuration.k"
4
5 module HASKELL-PREPROCESSING
```

```
6      imports HASKELL-SYNTAX
7      imports HASKELL-CONFIGURATION
8
9      //USER DEFINED LIST
10     //definition of ElemList
11
12     //syntax KItem ::= ElemList
13     syntax ElemList ::= List{Element, ",", ""} [strict]
14 //     syntax Int ::= lengthOfList(ElemList) [function]
15
16 //     rule lengthOfList(.ElemList) => 0
17 //     rule lengthOfList(val(K:K), L:ElemList) => 1 +Int
18 //     rule lengthOfList(valValue(K:K), L:ElemList) => 1 +Int
19 //     lengthOfList(L)
20
21     syntax Element ::= val(K) [strict]
22     syntax ElementResult ::= valValue(K)
23     syntax Element ::= ElementResult
24     syntax KResult ::= ElementResult
25     rule val(K:KResult) => valValue(K) [structural]
26
27     //form ElemList
28     //syntax ElemList ::= formElemList(K) [function]
29
30     //CONVERT ~> TO List
31     //list convert
32     //syntax List ::= convertToList(K) [function]
33     //rule convertToList(.K) => .List
34     //rule convertToList(A:KItem ~> B:K) => ListItem(A)
35     //convertToList(B)
36
37     syntax KItem ::= dealWithImports(K, K)
38
39     rule <k> 'modListSingle('module(A:K, , B:K)) =>
40     //     dealWithImports(A, B) ...</k>
41
42     (.Bag =>
43     <module>... //DOT DOT DOT MEANS OVERWRITE ONLY SOME
44     //     OF THE DEFAULTS
45     <moduleName> A </moduleName>
46     ...</module>
```

```
44     )
45
46     rule <k> 'modList ('module (A:K,, B:K),, C:K) =>
47         dealWithImports (A,B) ~> C ...</k>
48
49     (.Bag =>
50         <module>... //DOT DOT DOT MEANS OVERWRITE ONLY SOME
51         OF THE DEFAULTS
52         <moduleName> A </moduleName>
53         ...</module>
54     )
55
56 //     rule dealWithImports (Mod:K, A:K) => callInit (A)
57
58 //     rule <k> dealWithImports (Mod:K, A:K) => callInit (A) ...</k>
59 >
60
61 rule <k> dealWithImports (Mod:K, 'bodyimpandtop (A:K,, B:K))
62     => .K ...</k>
63
64     <importTree> L:List => L importListConvert (Mod, A) </
65     importTree>
66
67     <recurImportTree> L:List => L importListConvert (Mod, A)
68     </recurImportTree>
69
70     <moduleName> Mod </moduleName>
71     <imports> S:Set (.Set => SetItem(A)) </imports>
72     <moduleTempCode> OldTemp:K => B </moduleTempCode>
73
74 rule <k> dealWithImports (Mod:K, 'bodyimpdecls (A:K)) => .K
75     ...</k>
76
77     <importTree> L:List => L importListConvert (Mod, A) </
78     importTree>
79
80     <recurImportTree> L:List => L importListConvert (Mod, A)
81     </recurImportTree>
82
83     <moduleName> Mod </moduleName>
84     <imports> S:Set (.Set => SetItem(A)) </imports>
85
86 //     rule <k> dealWithImports (Mod:K, 'bodytopdecls (A:K)) =>
87     callInit (A) ...</k>
88
89     rule <k> dealWithImports (Mod:K, 'bodytopdecls (B:K)) => .K
90     ...</k>
```

```
76      <moduleName> Mod </moduleName>
77      <moduleTempCode> OldTemp:K => B </moduleTempCode>
78
79      //importlist convert
80      syntax List ::= importListConvert (K,K) [function]
81      syntax KItem ::= impObject (K,K)
82
83      rule importListConvert (Name:K, 'impDecls (A:K,, Rest:K)) =>
          importListConvert (Name, A) importListConvert (Name, Rest)
84      rule importListConvert ('moduleName (Name:K), 'impDecl (A:K,,
          Modid:K,, C:K,, D:K)) => ListItem (impObject (Name, Modid))
85      rule importListConvert (Name:K, .ImpDecls) => .List
```

---

## 5.1 Module Inclusion Tree

```
1      /*NEW TODO ALGORITHM
2  1. Construct tree for module inclusion
3  2. Check tree for cycles
4  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
5  (and specify scoping) (desugar the scope so that each type
      specifies the scope) */
6
7      syntax KItem ::= "checkImportCycle"
8      syntax KItem ::= "recurseImportTree"
9
10     /*      rule <k> performNextChecks
11              => checkUseVars
12              ~> (checkLabelUses
13              ~> (checkBlockAddress (.K)
14              ~> (checkNoNormalBlocksHavingLandingpad (.K, TNS
                  -Set TES)
15              ~> (checkAllExpBlocksHavingLandingpad (.K, TES)
16              ~> (checkAllExpInFromInvoke (.K, TES)
17              ~> (checkLandingpad
18              ~> checkLandingDomResumes)))))) ...</k> */
19
20     rule <k> startImportRecursion => checkImportCycle
21              ~> (recurseImportTree)...</
                k>
22
```

```
23   syntax KItem ::= cycleCheck(K,Map,List,List) [function] //
      current node, map of all nodes to visited or not, stack,
      graph
24   syntax Map ::= createVisitMap(List,Map) [function] //graph,
      visitmap
25   syntax KItem ::= getUnvisitedNode(K,K, Map) [function] //
      visitmap
26   syntax List ::= getNodeNeighbors(K,List) [function] //
      visitmap
27
28   rule <k> checkImportCycle
29       => cycleCheck(.K,createVisitMap(I, .Map),.List,I)
      ...</k>
30   <importTree> I:List </importTree>
31   <impTreeVMap> .Map => createVisitMap(I, .Map) </
      impTreeVMap>
32
33   syntax KItem ::= "visited"
34   syntax KItem ::= "unvisited"
35   syntax KItem ::= "none"
36
37   rule createVisitMap(ListItem(impObject(A:K,B:K)) Rest:List,
      M:Map)
38       => createVisitMap(Rest, M[A <- unvisited][B <-
      unvisited])
39   rule createVisitMap(.List, M:Map) => M
40
41   rule getUnvisitedNode(.K, .K, .Map) => none
42   rule getUnvisitedNode(.K, .K, (A:K |-> B:K) M:Map)
43       => getUnvisitedNode(A, B, M)
44   rule getUnvisitedNode(A:KItem, unvisited, M:Map) => A
45   rule getUnvisitedNode(A:KItem, visited, M:Map)
46       => getUnvisitedNode(.K, .K, M)
47
48
49
50   rule getNodeNeighbors(Node:K,.List) => .List
51   rule getNodeNeighbors(.K,Rest:List) => .List
52
53   rule getNodeNeighbors(Node:KItem,ListItem(impObject(Node,B:
      KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
      ListItem(B)
```

```
54     rule getNodeNeighbors(Node:KItem, ListItem(impObject(A:KItem,
55         B:KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
56         requires Node !=K A
57
58     rule cycleCheck(none, M:Map, .List, L:List) => .K
59     rule cycleCheck(.K, M:Map, .List, I:List) => cycleCheck(
60         getUnvisitedNode(.K, .K, M), M, .List, I)
61     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:List)
62         => cycleCheck(Node, M, S, I)
63     rule cycleCheck(Node:K, M:Map, S:List, I:List)
64         => cycleCheck(.K, M[Node <- visited],
65             getNodeNeighbors(Node,I) S, I)
66         requires Node !=K .K andBool Node !=K none
67     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
68         cycleCheck(Node, M, S, I)
69         requires S !=K .List
70
71     /*
72     rule cycleCheck(A:K, .K, .K, I:K) => cycleCheck(A,
73         createVisitMap(I, .Map), .List, I)
74
75     rule cycleCheck(Node:K, M:Map, S:List, I:K) => cycleCheck(.K
76         , M[Node <- visited], getNodeNeighbors(Node,I) S, I)
77
78     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
79         cycleCheck(Node, M, S, I)
80     //rule cycleCheck(.K, M:Map, .K, ListItem(impObject(A:K,B:K)
81         ) Rest:List) => cycleCheck(ListItem(impObject(A:K,B:K))
82         Rest:List)
83
84     */
```

---

## 5.2 Leaf DFS

```
1 //COPY IMPORT GRAPH, NEED SECOND GRAPH FOR RECURSING, ADDITIONAL
2   GRAPH FOR SELECTING IMPORTS FOR ALPHA* AND BETA*
3 //DFS for leaf
4 //acquire alpha and beta for leaf
5 //merge alpha and beta with imports to produce alpha* and beta*
6 //perform checks
```

```
6 //perform inferencing
7 //insert alpha* and beta* into importing modules
8 //remove all edges pointing to leaf
9
10 syntax KItem ::= "leafDFS"
11 syntax KItem ::= "getAlphaAndBeta"
12 syntax KItem ::= "getAlphaBetaStar"
13 syntax KItem ::= "performIndividualChecks"
14 syntax KItem ::= "performIndividualInferencing"
15 syntax KItem ::= "insertAlphaBetaStar"
16 syntax KItem ::= "removeAllEdges"
17 syntax KItem ::= "seeIfFinished"
18
19 rule <k> recurseImportTree => leafDFS
20                               ~> (getAlphaAndBeta
21                               //~> (getAlphaBetaStar
22                               ~> (
23                                   performIndividualInferencing
24                                   ))...</k>
25
26 //rule <k> dealWithImports (Mod:K, 'bodytopdecls (A:K)) =>
27 //    callInit (A) ...</k>
28 //
29 //    rule <k> leaf
30 //    => cycleCheck (.K, createVisitMap (I, .Map), .List, I)
31 //    ...</k>
32 //
33 //    <importTree> I:List </importTree>
34 //
35 //
36 //
37 //
38 //
39 //
40 //
41 //
42 //
43 //
44 //
45 //
46 //
47 //
48 //
49 //
50 //
51 //
52 //
53 //
54 //
55 //
56 //
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
68 //
69 //
70 //
71 //
72 //
73 //
74 //
75 //
76 //
77 //
78 //
79 //
80 //
81 //
82 //
83 //
84 //
85 //
86 //
87 //
88 //
89 //
90 //
91 //
92 //
93 //
94 //
95 //
96 //
97 //
98 //
99 //
100 //
101 //
102 //
103 //
104 //
105 //
106 //
107 //
108 //
109 //
110 //
111 //
112 //
113 //
114 //
115 //
116 //
117 //
118 //
119 //
120 //
121 //
122 //
123 //
124 //
125 //
126 //
127 //
128 //
129 //
130 //
131 //
132 //
133 //
134 //
135 //
136 //
137 //
138 //
139 //
140 //
141 //
142 //
143 //
144 //
145 //
146 //
147 //
148 //
149 //
150 //
151 //
152 //
153 //
154 //
155 //
156 //
157 //
158 //
159 //
160 //
161 //
162 //
163 //
164 //
165 //
166 //
167 //
168 //
169 //
170 //
171 //
172 //
173 //
174 //
175 //
176 //
177 //
178 //
179 //
180 //
181 //
182 //
183 //
184 //
185 //
186 //
187 //
188 //
189 //
190 //
191 //
192 //
193 //
194 //
195 //
196 //
197 //
198 //
199 //
200 //
201 //
202 //
203 //
204 //
205 //
206 //
207 //
208 //
209 //
210 //
211 //
212 //
213 //
214 //
215 //
216 //
217 //
218 //
219 //
220 //
221 //
222 //
223 //
224 //
225 //
226 //
227 //
228 //
229 //
230 //
231 //
232 //
233 //
234 //
235 //
236 //
237 //
238 //
239 //
240 //
241 //
242 //
243 //
244 //
245 //
246 //
247 //
248 //
249 //
250 //
251 //
252 //
253 //
254 //
255 //
256 //
257 //
258 //
259 //
260 //
261 //
262 //
263 //
264 //
265 //
266 //
267 //
268 //
269 //
270 //
271 //
272 //
273 //
274 //
275 //
276 //
277 //
278 //
279 //
280 //
281 //
282 //
283 //
284 //
285 //
286 //
287 //
288 //
289 //
290 //
291 //
292 //
293 //
294 //
295 //
296 //
297 //
298 //
299 //
300 //
301 //
302 //
303 //
304 //
305 //
306 //
307 //
308 //
309 //
310 //
311 //
312 //
313 //
314 //
315 //
316 //
317 //
318 //
319 //
320 //
321 //
322 //
323 //
324 //
325 //
326 //
327 //
328 //
329 //
330 //
331 //
332 //
333 //
334 //
335 //
336 //
337 //
338 //
339 //
340 //
341 //
342 //
343 //
344 //
345 //
346 //
347 //
348 //
349 //
350 //
351 //
352 //
353 //
354 //
355 //
356 //
357 //
358 //
359 //
360 //
361 //
362 //
363 //
364 //
365 //
366 //
367 //
368 //
369 //
370 //
371 //
372 //
373 //
374 //
375 //
376 //
377 //
378 //
379 //
380 //
381 //
382 //
383 //
384 //
385 //
386 //
387 //
388 //
389 //
390 //
391 //
392 //
393 //
394 //
395 //
396 //
397 //
398 //
399 //
400 //
401 //
402 //
403 //
404 //
405 //
406 //
407 //
408 //
409 //
410 //
411 //
412 //
413 //
414 //
415 //
416 //
417 //
418 //
419 //
420 //
421 //
422 //
423 //
424 //
425 //
426 //
427 //
428 //
429 //
430 //
431 //
432 //
433 //
434 //
435 //
436 //
437 //
438 //
439 //
440 //
441 //
442 //
443 //
444 //
445 //
446 //
447 //
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825 //
826 //
827 //
828 //
829 //
830 //
831 //
832 //
833 //
834 //
835 //
836 //
837 //
838 //
839 //
840 //
841 //
842 //
843 //
844 //
845 //
846 //
847 //
848 //
849 //
850 //
851 //
852 //
853 //
854 //
855 //
856 //
857 //
858 //
859 //
860 //
861 //
862 //
863 //
864 //
865 //
866 //
867 //
868 //
869 //
870 //
871 //
872 //
873 //
874 //
875 //
876 //
877 //
878 //
879 //
880 //
881 //
882 //
883 //
884 //
885 //
886 //
887 //
888 //
889 //
890 //
891 //
892 //
893 //
894 //
895 //
896 //
897 //
898 //
899 //
900 //
901 //
902 //
903 //
904 //
905 //
906 //
907 //
908 //
909 //
910 //
911 //
912 //
913 //
914 //
915 //
916 //
917 //
918 //
919 //
920 //
921 //
922 //
923 //
924 //
925 //
926 //
927 //
928 //
929 //
930 //
931 //
932 //
933 //
934 //
935 //
936 //
937 //
938 //
939 //
940 //
941 //
942 //
943 //
944 //
945 //
946 //
947 //
948 //
949 //
950 //
951 //
952 //
953 //
954 //
955 //
956 //
957 //
958 //
959 //
960 //
961 //
962 //
963 //
964 //
965 //
966 //
967 //
968 //
969 //
970 //
971 //
972 //
973 //
974 //
975 //
976 //
977 //
978 //
979 //
980 //
981 //
982 //
983 //
984 //
985 //
986 //
987 //
988 //
989 //
990 //
991 //
992 //
993 //
994 //
995 //
996 //
997 //
998 //
999 //
1000 //
```



```
41     rule returnLeafDFS(.K,ListItem(impObject(Node:KItem,B:KItem)
42       ) I:List,M:Map) => returnLeafDFS(B,I,M)
43     rule returnLeafDFS(Node:KItem,I:List,M:Map) => returnLeafDFS
44       (innerLeafDFS(Node,I),I,M)
45       requires innerLeafDFS(Node,I) !=K none
46     rule returnLeafDFS(Node:KItem,I:List,M:Map) => loadModule(
47       Node)
48       requires innerLeafDFS(Node,I) ==K none
49     rule innerLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem
50       )) I:List) => B
51     rule innerLeafDFS(Node:KItem,ListItem(impObject(A:KItem,B:
52       KItem)) I:List) => innerLeafDFS(Node,I)
53     requires Node !=K A
54     rule innerLeafDFS(Node:KItem,.List) => none
55 //     returnLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem))
56 //       I:List,M:Map) => returnLeafDFS(B,I,M)
57 //
58 //
59 //
60 //
61 //
62 //
63 //
64 //
65 //
66 //
67 //
```

---

### 5.3 Insert AlphaBetaStar

```
1 //     syntax KItem ::= "insertAlphaBetaStar"
2
3     syntax KItem ::= insertABRec(K,List)
4     syntax KItem ::= insertAB(K)
5
```

```
6      rule <k> insertAlphaBetaStar => insertABRec (Mod, Imp) ...</k>
      >
7      <tempModule> Mod:KItem </tempModule>
8      <importTree> Imp:List </importTree>
9
10     rule <k> insertABRec (Node:KItem, ListItem (impObject (B:KItem,
      Node)) I:List) => insertAB (B) ~> insertABRec (Node, I)
      ...</k>
11
12     rule <k> insertABRec (Node:KItem, ListItem (impObject (B:KItem,
      C:KItem)) I:List) => insertABRec (Node, I) ...</k>
13         requires Node !=K C
14
15     rule <k> insertAB (B) => .K ...</k>
16
17     <tempAlphaStar> Alph:KItem </tempAlphaStar>
18     <tempBetaStar> Bet:KItem </tempBetaStar>
19
20     <moduleName> 'moduleName (B) </moduleName>
21     <moduleImpAlphas> ImpAlphas:List => ListItem (Alph)
      ImpAlphas </moduleImpAlphas>
22     <moduleImpBetas> ImpBetas:List => ListItem (Bet)
      ImpBetas </moduleImpBetas>
23
24
25 endmodule
```

---

# CHAPTER 6

## INFERRNCING

### 6.1 Data Structures

The next step is the actual type inference algorithm. I needed to create a syntax for polymorphic types that may contain monomorphic type variables and polymorphic type variables. Then I made a map from type constructor names to arities, called Delta. Then I made a map from data constructors and term identifiers to their most general polymorphic types, called beta. The first part was converting the T data structure into beta, which is more suited for type inferencing. Something to note is that 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 just simply runs forever. This is unlike the OCaml semantics, which does not allow for mutually recursive functions.

Another thing to note is that the

---

1 [context =>]

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

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

#### 6.1.1 Transform T into Beta

```
1      // STEP 3 Transform T into beta
2
3      syntax KItem ::= "startTTransform"
4      syntax KItem ::= "constructDelta"
5      syntax KItem ::= "constructBeta"
6
7      rule <k> startTTransform
8          => constructDelta
9          ~> (constructBeta) ...</k>
10
11     rule <k> constructDelta
12         => makeDelta(T,.Map) ...</k>
13         <tempT> T:K </tempT>
14
15     syntax KItem ::= makeDelta(K,Map) [function] //(T,Delta)
16     syntax KItem ::= newDelta(Map) //Delta
17     syntax KItem ::= newBeta(Map) //beta
18     syntax List ::= retPolyList(K,List) [function] //(T,Delta)
```

---

### Construct Delta

```
1      rule makeDelta(TList(ListItem(TObject(ModName:K,A:K,Polys:K,
2          C:K)) Rest:List),M:Map) =>
3          makeDelta(TList(Rest),M[ModPlusType(ModName,A) <- size(
4              retPolyList(Polys,.List))])
5      rule makeDelta(TList(.List),M:Map) => newDelta(M)
6
7      rule retPolyList('typeVars(A:K,,Rest:K),NewList:List) =>
8          retPolyList(Rest, ListItem(A) NewList)
9      rule retPolyList(.TyVars,L:List) => L
10
11     rule <k> newDelta(M:Map)
12         => .K ...</k>
13     <tempDelta> OldDelta:K => M </tempDelta>
```

---

### Construct Beta

```
1      rule <k> constructBeta
2          => makeBeta(T,.Map) ...</k>
3      <tempT> T:K </tempT>
4
5      syntax KItem ::= makeBeta(K,Map) [function] //(T,Beta,Delta)
```

```

6
7   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) =>
8       makeBeta (TList (ListItem (TObject (ModName, A, B, InnerRest))
      Rest), Beta [ModPlusType (ModName, Con) <- betaParser (E
      , B, A)])
9   rule makeBeta (TList (ListItem (TObject (ModName:K, A:K, B:K, .List
      )) Rest:List), Beta:Map) =>
10       makeBeta (TList (Rest), Beta)
11   rule makeBeta (TList (.List), Beta:Map) =>
12       newBeta (Beta)
13 //   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) =>
14 //       makeBeta (TList (ListItem (TObject (ModName, A, B, InnerRest
      )) Rest), Beta)
15
16   syntax KItem ::= betaParser (K, K, K) [function] // (Tree Piece,
      NewSyntax, Parameters, Constr)
17   syntax Set ::= getTyVarsRHS (K, List) [function]
18
19   syntax KItem ::= forAll (Set, K)
20   syntax KItem ::= funtype (K, K)
21
22   syntax Set ::= listToSet (List, Set) [function]
23
24   rule listToSet (ListItem (A:KItem) L:List, S:Set) => listToSet
      (L, SetItem (A) S)
25   rule listToSet (.List, S:Set) => S
26
27
28 //if optbangAtypes, need to see if first variable is a typecon
29 //if its a typecon then need to go into Delta and see the amount
      of parameters it has
30 //then count the number of optbangAtypes after the typecon
31   rule betaParser ('constrCon (A:K,, B:K), Par:K, Con:K) =>
      forAll (getTyVarsRHS (B, .List), betaParser (B, Par, Con))
32   rule betaParser ('optBangATypes ('optBangAType ('emptyBang (.
      KList),, 'atypeTyVar (Tyv:K)),, Rest:K), Par:K, Con:K) =>
      funtype (Tyv, betaParser (Rest, Par, Con))
33   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))
34 rule betaParser('optBangATypes('optBangAType('emptyBang(.
    KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K, Con:K) =>
    funtype(Tyc, betaParser(Rest, Par, Con))
35 rule betaParser(.OptBangATypes, Par:K, Con:K) => '
    simpleTypeCon(Con,, Par)
36 // rule betaParser('optBangATypes('optBangAType('emptyBang(.
    KList),, 'atypeGTyCon(Tyc:K)),, Rest:KItem)) => getTypeVars(A
    ) getTypeVars(Rest)
37 // rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K
    )) => getTypeVars(Rest)
38 // rule getTypeVars('atypeGTyCon(A:K)) => .List
39 // rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
40 // rule getTypeVars(.OptBangATypes) => .List
41
42 rule getTyVarsRHS(.OptBangATypes,Tylist:List) => listToSet(
    Tylist, .Set)
43
44 rule <k> newBeta(M:Map)
45     => .K ...</k>
46     <tempBeta> OldBeta:K => M </tempBeta>
```

---

## 6.2 Inferencing Rules

Haskell is a strong and static type system.

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

Type inferencing

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

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

## 6.3 Type theory

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

## 6.4 Lambda Calculus

The Lambda Calculus was created to

## 6.5 Hindley-Milner

## 6.6 Definition of Substitution

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

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

## 6.7 Composition of Substitutions

## 6.8 Inferencing Algorithm

$$\begin{array}{c} \frac{}{\Gamma \vdash c : \tau \mid \text{unify}\{(\tau, \text{freshInstance}(\tau))\}} \text{ Constant} \\[1em] \frac{}{\Gamma \vdash x : \tau \mid \text{unify}\{(\tau, \text{freshInstance}(\Gamma(x)))\}} \text{ Variable} \\[1em] \frac{[x : \tau_1] + \Gamma \vdash e : \tau_2 \mid \sigma}{\Gamma \vdash \lambda x \rightarrow e : \tau \mid \text{unify}\{(\sigma(\tau), \sigma(\tau_1 \rightarrow \tau_2))\} \circ \sigma} \text{ Lambda} \end{array}$$

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

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

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

[cs 421 gather exp ty substitution mp]

```

1  requires "haskell-syntax.k"
2  requires "haskell-configuration.k"
3  requires "haskell-preprocessing.k"
4
5  module HASKELL-TYPE-INFERENCING
6      imports HASKELL-SYNTAX
7      imports HASKELL-CONFIGURATION
8      imports HASKELL-PREPROCESSING
9
10     syntax KItem ::= "Bool" //Boolean
11
12     // STEP 4 Type Inferencing
13     syntax KItem ::= inferenceShell(K) [function]//Input,
14         AlphaMap, Beta, Delta, Gamma
15     //syntax KItem ::= typeInferenceFun(K,Map,Map,Map,Map,K,K) [
16         function]//Input, Alpha, Beta, Delta, Gamma
17     //syntax KItem ::= typeInferenceFun(Map,K,K) //Gamma,
18         Expression, Guessed Type
19     syntax Map ::= genGamma(K,Map,K) [function] //Apatlist,
20         Gamma Type
21     syntax KItem ::= genLambda(K,K) [function]
22     syntax KItem ::= guessType(Int)
23 //     syntax KItem ::= lambdaReturn(K,K,K)
24     syntax KItem ::= freshInstance(K, Int) [function]
25     syntax Int ::= paramSize(K) [function]
26
27     syntax KItem ::= mapBag(Map)
28     syntax KResult ::= mapBagResult(Map)
29
30     syntax Map ::= gammaSub(Map,Map,Map) [function]//
31         substitution, gamma

```



```
29     rule <k> performIndividualInferencing => inferenceShell(Code
30         ) ...</k>
31         <tempModule> Mod:KItem </tempModule>
32         <moduleName> 'moduleName (Mod) </moduleName>
33         <moduleTempCode> Code:KItem </moduleTempCode>
34
35     rule inferenceShell('topdeclslist('type(A:K,, B:K),, Rest:K)
36         ) =>
37         inferenceShell(Rest) //constructalpha
38     rule inferenceShell('topdeclslist('data(A:K,, B:K,, C:K,, D:
39         K),, Rest:K)) =>
40         inferenceShell(Rest)
41     rule inferenceShell('topdeclslist('newtype(A:K,, B:K,, C:K,,
42         D:K),, Rest:K)) =>
43         inferenceShell(Rest)
44     rule inferenceShell('topdeclslist('class(A:K,, B:K,, C:K,, D
45         :K),, Rest:K)) =>
46         inferenceShell(Rest)
47     rule inferenceShell('topdeclslist('instance(A:K,, B:K,, C:K
48         ,, D:K),, Rest:K)) =>
49         inferenceShell(Rest)
50     rule inferenceShell('topdeclslist('default(A:K,, B:K,, C:K,,
51         D:K),, Rest:K)) =>
52         inferenceShell(Rest)
53     rule inferenceShell('topdeclslist('foreign(A:K,, B:K,, C:K,,
54         D:K),, Rest:K)) =>
55         inferenceShell(Rest)
56     rule inferenceShell('topdeclslist('topdecldecl(A:K),, Rest:K
57         )) =>
58         typeInferenceFun(.ElemList, .Map,A,guessType(0)) ~>
59         inferenceShell(Rest)
60
61     rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
62         declFunLhsRhs(Fn:K,, Lhsrhs:K), Guess:K) =>
63         typeInferenceFun(.ElemList, Gamma, Lhsrhs, Guess) ...</
64         k>
65     rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
66         eqExpOptDecls(Ex:K,, Optdecls:K), Guess:K) =>
67         typeInferenceFun(.ElemList, Gamma, Ex, Guess) ...</k>
```

---

### 6.8.1 Variable Rule

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

```

1      //T-App
2      //rule typeInferenceFun('aexpQVar(Var:K), Alpha:Map, Beta:
      Map, Delta:Map, (Var |-> Sigma:K) Gamma:Map, .K, .K) =>
      Sigma
3      //Gamma Proves x:phi(tau) if Gamma(x) = \forall alpha_1,
      ..., alpha_n . tau
4      //where phi replaces all occurrences of alpha_1, ...,
      alpha_n by monotypes tau_1, ..., tau_n
5
6      rule <k> typeInferenceFun(.ElemList, (Var |-> Type:K) Gamma:
      Map, 'aexpQVar(Var:K), Guess:KItem)
7      => mapBagResult (uniFun (ListItem (uniPair (Guess,
      freshInstance (Type, TypeIt)))) ...</k> //Variable
      rule
8      <typeIterator> TypeIt:Int => TypeIt +Int paramSize (Type
      ) </typeIterator>

```

---

### 6.8.2 Constant Rule

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

```

1
2      rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'aexpGCon('
      conTyCon (Mid:K,, Gcon:K)), Guess:KItem)
3      => mapBagResult (uniFun (ListItem (uniPair (Guess,
      freshInstance (Type, TypeIt)))) ...</k> //Constant
      rule
4      <tempBeta> (ModPlusType (Mid, Gcon) |-> Type:K) Beta:Map
      </tempBeta>
5      <typeIterator> TypeIt:Int => TypeIt +Int paramSize (Type
      ) </typeIterator>

```

---

### 6.8.3 Lambda Rule

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

```

1
2   syntax KItem ::= typeInferenceFun(ElemList, Map, K, K) [
      strict(1)]
3   syntax KItem ::= typeInferenceFunLambda(ElemList, K, K, K) [
      strict(1)]
4   /* automatically generated by the strict(1) in typeInferenceFun
      or typeInferenceFunAux
5   rule typeInferenceFunAux(Es:ElemList, C:K, A:K, B:K) => Es ~>
      typeInferenceFun(HOLE, C, A, B)
6       requires notBool isKResult(Es)
7   rule Es:KResult ~> typeInferenceFunAux(HOLE, C:K,A:K, B:K) =>
      typeInferenceFun(Es, C, A, B)
8   */
9
10  //lambda rule
11  rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'lambdaFun(
      Apatlist:K,, Ex:K), Guess:KItem)
12      => 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>
13      <typeIterator> TypeIt:Int => TypeIt +Int 2 </
      typeIterator>
14
15  rule <k> typeInferenceFunLambda(valValue(mapBagResult(Sigma:
      Map)), .ElemList, Tau:K, Tauone:K, Tautwo:K)
16      => mapBagResult(compose(uniFun(ListItem(uniPair(typeSub
      (Sigma,Tau),typeSub(Sigma,funtime(Tauone,Tautwo))))
      ,Sigma)) ...</k>

```

---

#### 6.8.4 Application Rule

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

```

1
2   syntax KItem ::= typeInferenceFunAppli(ElemList, Map, K, K,
      Map) [strict(1)]
3
4   //application rule

```

```

5     rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'funApp(Eone
      :K,, Etwo:K), Guess:KItem)
6       => typeInferenceFunAppli(val(typeInferenceFun(.
          ElemList, Gamma, Eone, funtype(guessType(TypeIt),
          Guess))), .ElemList, Gamma, Etwo, guessType(TypeIt)
          , .Map) ...</k>
7     <typeIterator> TypeIt:Int => TypeIt +Int 1 </
      typeIterator>
8
9     rule <k> typeInferenceFunAppli(valValue(mapBagResult(
      Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem,
      guessType(TypeIt:Int), .Map)
10      => typeInferenceFunAppli(val(typeInferenceFun(.
          ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo,
          typeSub(Sigmaone, guessType(TypeIt)))), .ElemList,
          .Map, .K, .K, Sigmaone) ...</k>
11
12     rule <k> typeInferenceFunAppli(valValue(mapBagResult(
      Sigmatwo:Map)), .ElemList, .Map, .K, .K, Sigmaone:Map)
13      => mapBagResult(compose(Sigmatwo, Sigmaone)) ...</k>

```

---

### 6.8.5 IfThenElse Rule

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

```

1     syntax KItem ::= typeInferenceFunIfThen(ElemList, Map, K, K,
      K, Map, Map) [strict(1)]
2
3     //if_then_else rule
4     rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'ifThenElse(
      Eone:K,, Optsem:K,, Etwo:K,, Optsemtwo:K,, Ethree:K),
      Guess:KItem)
5       => typeInferenceFunIfThen(val(typeInferenceFun(.
          ElemList, Gamma, Eone, Bool)), .ElemList, Gamma,
          Etwo, Ethree, Guess, .Map, .Map) ...</k>
6
7     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
      Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem, Ethree:
      KItem, Guess:KItem, .Map, .Map)

```

```

8      => typeInferenceFunIfThen(val(typeInferenceFun(.
      ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo,
      typeSub(Sigmaone, Guess))), .ElemList, Gamma, .K,
      Ethree, Guess, Sigmaone, .Map) ...</k>
9
10     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
      Sigmatwo:Map)), .ElemList, Gamma:Map, .K, Ethree:KItem,
      Guess:KItem, Sigmaone:Map, .Map)
11       => typeInferenceFunIfThen(val(typeInferenceFun(.
      ElemList, gammaSub(compose(Sigmatwo, Sigmaone),
      Gamma, .Map), Ethree, typeSub(compose(Sigmatwo,
      Sigmaone), Guess))), .ElemList, .Map, .K, .K, .K,
      Sigmaone, Sigmatwo) ...</k>
12
13     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
      Sigmathree:Map)), .ElemList, .Map, .K, .K, .K, Sigmaone:
      Map, Sigmatwo:Map)
14       => mapBagResult(compose(compose(Sigmathree, Sigmatwo),
      Sigmaone)) ...</k>

```

---

### 6.8.6 LetIn Rule

The LetIn Rule must allow for mutual recursion. This means that an example program that must be inferred is

```

1 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  $f$ .

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

```

1
2     syntax KItem ::= typeInferenceFunLetIn(ElemList, Map, Map, K
      , K, K, Int, Int, Map, Map) [strict(1)]
3     syntax KItem ::= grabLetDeclName(K, Int) [function]
4     syntax KItem ::= grabLetDeclExp(K, Int) [function]
5     syntax KItem ::= mapLookup(Map, K) [function]
6     syntax Map ::= makeDeclMap(K, Int, Map) [function]
7     syntax Map ::= applyGEN(Map, Map, Map, Map) [function]
8

```

```

9      //Haskell let in rule (let rec in exp + let in rule combined
      )
10     //gamma |- let rec f1 = e1 and f2 = e2 and f3 = e3 .... in e
      =>
11     //beta, [f1 -> tau1, f2 -> tau2, f3 -> tau3,...] + gamma |-
      e1 : tau1 | sigma1, [f1 -> sigma1(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
12     rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'letIn(D:K,,
      E:K), Guess:KItem)
13       => typeInferenceFunLetIn(.ElemList, Gamma, makeDeclMap
      (D, TypeIt, .Map), D, E, Guess, 0, TypeIt, .Map,
      Beta) ...</k>
14     <typeIterator> TypeIt:Int => TypeIt +Int size(
      makeDeclMap(D, TypeIt, .Map)) </typeIterator>
15     <tempBeta> Beta:Map </tempBeta>
16
17     rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap
      :Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int
      , OldSigma:Map, Beta:Map)
18       => 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>
19     //=> typeInferenceFunLetIn(val(typeInferenceFun(
      DeclMap, grabLetDeclExp(D, Iter +Int TypeIt), Guess
      )), .ElemList, Gamma, DeclMap, D, E, Guess, Iter,
      TypeIt, OldSigma) ...</k>
20     requires Iter <Int (size(DeclMap))
21
22     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)
23       => typeInferenceFunLetIn(.ElemList, gammaSub(Sigma,
      Gamma,.Map), gammaSub(Sigma, DeclMap,.Map), D, E,
      typeSub(Sigma, Guess), Iter +Int 1, TypeIt, compose

```

```

      (Sigma, OldSigma), Beta) ...</k>
24     requires Iter <Int (size(DeclMap))
25
26     rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap
      :Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int
      , OldSigma:Map, Beta:Map)
27       => typeInferenceFunLetIn(val(typeInferenceFun(.
      ElemList, Gamma applyGEN(Gamma, DeclMap, .Map, Beta
      ), E, Guess)), .ElemList, Gamma, DeclMap, D, E,
      Guess, Iter, TypeIt, OldSigma, Beta) ...</k>
28     requires Iter >=Int (size(DeclMap))
29
30     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)
31       => mapBagResult(compose(Sigma, OldSigma)) ...</k>
32     requires Iter >=Int (size(DeclMap))
```

---

```

1
2     rule mapLookup((Name |-> Type:KItem) DeclMap:Map, Name:KItem
      ) => Type
3     rule mapLookup(DeclMap:Map, Name:KItem) => Name
4       requires notBool(Name in keys(DeclMap))
5
6     rule makeDeclMap('decls(Dec:K), TypeIt:Int, NewMap:Map) =>
      makeDeclMap(Dec, TypeIt, NewMap)
7     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)])
8     rule makeDeclMap(.DeclList, TypeIt:Int, NewMap:Map) =>
      NewMap
9
10    rule grabLetDeclName('decls(Dec:K), Iter:Int) =>
      grabLetDeclName(Dec, Iter)
11    rule grabLetDeclName('declsList(Dec:K,, Rest:K), Iter:Int)
      => grabLetDeclName(Rest, Iter -Int 1)
12      requires Iter >Int 0
13    rule grabLetDeclName('declsList('declPatRhs('apatVar(Var:K)
      ,, Righthand:K),, Rest:K), Iter:Int) => Var
14      requires Iter <=Int 0
15
```

```
16
17   rule grabLetDeclExp('decls(Dec:K), Iter:Int) =>
      grabLetDeclExp(Dec, Iter)
18   rule grabLetDeclExp('declsList(Dec:K,, Rest:K), Iter:Int) =>
      grabLetDeclExp(Rest, Iter -Int 1)
19       requires Iter >Int 0
20   rule grabLetDeclExp('declsList('declPatRhs('apatVar(Var:K),,
      Righthand:K),, Rest:K), Iter:Int) => grabLetDeclExp(
      Righthand, Iter)
21       requires Iter <=Int 0
22   rule grabLetDeclExp('eqExpOptDecls(Righthand:K,, Opt:K),
      Iter:Int) => 'eqExpOptDecls(Righthand,, Opt)
23
24   rule genGamma('apatVar(Vari:K), Gamma:Map, Guess:K) => Gamma
      [Vari <- Guess]
25   rule genGamma('apatCon(Vari:K,, Pattwo:K), Gamma:Map, Guess:
      K) => Gamma[Vari <- Guess]
26
27   rule genLambda('apatVar(Vari:K), Ex:K) => Ex
28   rule genLambda('apatCon(Vari:K,, Pattwo:K), Ex:K) => '
      lambdaFun(Pattwo,, Ex)
29
30
31   rule gammaSub(Sigma:Map, (Key:KItem |-> Type:KItem) Gamma:
      Map, Newgamma:Map)
32       => gammaSub(Sigma, Gamma, Newgamma[Key <- typeSub(Sigma,
      Type) ] )
33
34   rule gammaSub(Sigma:Map, .Map, Newgamma:Map)
35       => Newgamma
36
37   rule freshInstance(guessType(TypeIt:Int), Iter:Int) =>
      guessType(TypeIt)
38   rule freshInstance(forAll(.Set, B:K), Iter:Int) => B
39   rule freshInstance(forAll(SetItem(C:KItem) A:Set, B:K), Iter
      :Int) => freshInstance(forAll(A, freshInstanceInner(C, B,
      Iter)), Iter +Int 1)
40
41   syntax KItem ::= freshInstanceInner(K,K,Int) [function]
42
43   rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K), Iter:
      Int) => funtype(freshInstanceInner(Repl,A,Iter),
      freshInstanceInner(Repl,B,Iter))
```



```
44     rule freshInstanceInner(Repl:KItem, Repl, Iter:Int) =>
        guessType(Iter)
45     rule freshInstanceInner(Repl:KItem, Target:KItem, Iter:Int)
        => Target [owise]
46
47     rule paramSize(forAll(A:Set, B:K)) => size(A)
48     rule paramSize(A:K) => 0 [owise]
49
50
51     rule applyGEN(Gamma:Map, (Key:KItem |-> Type:KItem) DeclMap
        :Map, NewMap:Map, Beta:Map)
52     => applyGEN(Gamma, DeclMap, NewMap[Key <- gen(Gamma, Type
        , Beta)], Beta)
53
54     rule applyGEN(Gamma:Map, .Map, NewMap:Map, Beta:Map)
55     => NewMap
56
57     //GEN
58     //GEN(Gamma, Tau) => Forall alpha
59
60     syntax KItem ::= gen(Map, K, Map) [function]
61     syntax Set ::= freeVarsTy(K, Map) [function]
62     syntax Set ::= freeVarsEnv(Map, Map) [function]
63
64
65     rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
        => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
        freeVarsEnv(Gamma, Beta), Tau)
66     rule gen(Gamma:Map, Tau:KItem, Beta:Map) => forAll(
        freeVarsTy(Tau, Beta) -Set freeVarsEnv(Gamma, Beta), Tau)
        [owise]
67
68     //rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
        => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
        freeVarsEnv(Gamma, Beta), Tau)
69
70     rule freeVarsTy(guessType(TypeIt:Int), Beta:Map) => SetItem(
        guessType(TypeIt:Int))
71     rule freeVarsTy(funtype(Tauone:KItem, Tautwo:KItem), Beta:
        Map) => freeVarsTy(Tauone, Beta) freeVarsTy(Tautwo, Beta)
72     rule freeVarsTy(Tau:KItem, Beta:Map) => .Set
73     requires (forAll(.Set, Tau)) in values(Beta)
```

```

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

```

---

## 6.9 Unification Algorithm

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

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

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

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

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

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

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

Let  $\phi = x \mapsto t$

Let  $\psi = \text{Unif}(\phi(S'))$

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

Note:  $x \mapsto a \circ y \mapsto b = y \mapsto (x \mapsto a(b)) \circ x \mapsto a$  if y not in a

[cs 421 class notes]

```

1     //Unification
2
3     syntax Map ::= uniFun(List) [function]
4     syntax Bool ::= isVarType(K) [function]
5     syntax Bool ::= notChildVar(K,K) [function]
6     syntax KItem ::= uniPair(K,K)
7
8     syntax List ::= uniSub(Map,K) [function] //apply
        substitution to unification
9
10    syntax KItem ::= typeSub(Map,K) [function] //apply
        substitution to type
11    syntax Map ::= compose(Map,Map) [function]
12
13    rule uniFun(.List) => .Map //substi(.K,.K) is id
        substitution

```

```
14
15     rule uniFun(ListItem(uniPair(S:K,S)) Rest:List) => uniFun(
16         Rest) //delete rule
17
18     rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List) => uniFun(
19         ListItem(uniPair(T,S)) Rest) //orient rule
20         requires isVarType(T) andBool (notBool isVarType(S))
21
22     rule uniFun(ListItem(uniPair(funtype(A:K, B:K), funtype(C:K,
23         D:K))) Rest:List) => uniFun(ListItem(uniPair(A, C))
24         ListItem(uniPair(B, D)) Rest:List) //decompose rule
25         function type
26
27     rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List)
28         => compose((S |-> typeSub(uniFun(uniSub((S |-> T),Rest)),T
29             )),uniFun(uniSub((S |-> T),Rest))) //eliminate rule
30
31     // => compose(uniFun(uniSub((S |-> T),Rest)),(S |-> typeSub
32         (uniFun(uniSub((S |-> T),Rest)),T))) //eliminate rule
33         requires isVarType(S) andBool notChildVar(S,T)
34
35     rule isVarType(S:K) => true
36         requires getKLabel(S) ==KLabel 'guessType
37
38     rule isVarType(S:K) => false [owise]
39
40
41     rule notChildVar(S:K,T:K) => true
42
43
44     rule uniSub(Sigma:Map,.List) => .List
45
46     rule uniSub(.Map,L:List) => L
47
48     rule uniSub(Sigma:Map, Rest:List ListItem(uniPair(A:K, B:K))
49         ) => uniSub(Sigma, Rest) ListItem(uniPair(typeSub(Sigma,
50             A), typeSub(Sigma, B)))
51
52
53     //rule typeSub(substi(.Map),Tau:KItem) => Tau
54
55     rule typeSub(Sigma:Map (Tau |-> Newtau:KItem),Tau:KItem) =>
56         typeSub(Sigma (Tau |-> Newtau),Newtau)
57
58     rule typeSub(Sigma:Map,funtype(Tauone:KItem,Tautwo:KItem))
59         => funtype(typeSub(Sigma,Tauone),typeSub(Sigma,Tautwo))
60
61     rule typeSub(Sigma:Map,Tau:KItem) => Tau [owise]
62
63
64     syntax Map ::= composeIn(Map, Map, Map, K, K) [function]
65
66
67     rule compose(Sigmaone:Map, Sigmatwo:Map) => composeIn(
68         Sigmaone, Sigmatwo, .Map, .K, .K)
```

```
45
46   rule composeIn(Sigmaone:Map, (Key:KItem |-> Type:KItem)
      Sigmatwo:Map, NewMap:Map, .K, .K) => composeIn(Sigmaone,
      Sigmatwo, NewMap, Key, Type)
47
48   rule composeIn((Keyone |-> Typetwo:KItem) Sigmaone:Map,
      Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) =>
      composeIn(Sigmaone, Sigmatwo, NewMap, Keyone, Typeone)
49
50   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)
51     requires notBool(Keyone in keys(Sigmaone))
52
53   rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
      Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
      Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]
54
55   rule composeIn(Sigmaone:Map, .Map, NewMap:Map, .K, .K) =>
      Sigmaone NewMap
56 endmodule
```

---

# CHAPTER 7

## CONCLUSION

This project helped me with my K knowledge. I also learned a lot about how compilers work and how programming languages are defined. Also, 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

```
1 // Syntax from haskell 2010 Report
2 // https://www.haskell.org/onlinereport/haskell2010/haskellch10.
   html#x17-17500010
3
4 module HASKELL-SYNTAX
5
6     syntax Integer ::= Token{ ([0-9]+)
7         | (([0][o]|[0][O])[0-7]+)
8         | (([0][x] | [0][X])[0-9a-fA-F]+) } [onlyLabel]
9
10    syntax CusFloat ::= Token{ ([0-9]+[\.][0-9]+([e E
11        ][\+|-]?[0-9]+)?)
12        | ([0-9]+[e E][\+|-]?[0-9]+) } [
13            onlyLabel]
14
15    syntax CusChar ::= Token{ [\'] (~[\'\\&]) [\'] } [onlyLabel]
16    syntax CusString ::= Token{ [\" ] (~[\"\\&]*) [\" ] } [onlyLabel]
17    syntax VarId ::= Token{ [a-z\_][a-z A-Z\_0-9\']* } [onlyLabel]
18        | "size" [onlyLabel]
19    syntax ConId ::= Token{ [A-Z][a-zA-Z\_0-9\']* } [onlyLabel]
20    syntax VarSym ::= Token{
21        ([\! \# \$ \% \& \* \+ \/ \> \? \^][\! \# \$ \% \& \* \+ \.
22            \/ \< \= \> \? \@ \\\ \^ \\\ \- \~ \:]* )
23        | [\-] | [\.]
24        | ([\.] [\! \# \$ \% \& \* \+ \/ \< \= \> \? \@ \\\ \^ \\\ \- \~
25            \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\\ \^ \\\ \-
26            \~ \:]* )
27        | ([\-][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\\ \^ \\\ \-
28            \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\\ \^ \\\ \-
29            \:]* )
30        | ([\@][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\\ \^ \\\ \-
31            \~ \:]* )
32        | ([\~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\\ \^ \\\ \-
33            \~ \:]* )
34    }
```

```

23 | ([\\][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~ \\:]+)
24 | ([\\|][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~ \\:]+)
25 | ([\\:][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~ \\:]*)
26 | ([\\<][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\~
    \\:][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~ \\:]*)
27 | ([\\=][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\? \\@ \\ \\ ^ \\| \\~
    \\:][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\< \\= \\> \\? \\@ \\ \\ ^ \\| \\-
    \\~ \\:]*)*} [onlyLabel]
28 syntax ConSym ::= Token{[\\:][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\<
    \\= \\> \\? \\@ \\ \\ ^ \\| \\- \\~][\\! \\# \\$ \% \\& \\* \\+ \\. \\/ \\<
    \\= \\> \\? \\@ \\ \\ ^ \\| \\- \\~ \\:]*} [onlyLabel]
29
30 syntax IntFloat ::= "(" Integer ")" [bracket] //NOT
    OFFICIAL SYNTAX
31 | "(" CusFloat ")" [bracket]
32 syntax Literal ::= IntFloat | CusChar | CusString
33 syntax TyCon ::= ConId
34 syntax ModId ::= ConId | ConId "." ModId [klabel('conModId)
    ]
35 syntax QTyCon ::= TyCon | ModId "." TyCon [klabel('conTyCon)
    ]
36 syntax QVarId ::= VarId | ModId "." VarId [klabel('qVarIdCon
    )]
37 syntax QVarSym ::= VarSym | ModId "." VarSym [klabel('
    qVarSymCon)]
38 syntax QConSym ::= ConSym | ModId "." ConSym [klabel('
    qConSymCon)]
39 syntax TyVars ::= List{TyVar, ""} [klabel('typeVars)] //used
    in SimpleType syntax
40 syntax TyVar ::= VarId
41 syntax TyVarTuple ::= TyVar "," TyVar [klabel('
    twoTypeVarTuple)]
42 | TyVar "," TyVarTuple [klabel('
    typeVarTupleCon)]
43
44 syntax Con ::= ConId | "(" ConSym ")" [klabel('
    conSymBracket)]

```

```
45     syntax Var ::= VarId | "(" VarSym ")" [klabel('
        varSymBracket)]
46     syntax QVar ::= QVarId | "(" QConSym ")" [klabel('
        qVarBracket)]
47     syntax QCon ::= QTyCon | "(" GConSym ")" [klabel('
        gConBracket)]
48
49     syntax QConOp ::= GConSym | "\"" QTyCon "\"" [klabel('
        qTyConQuote)]
50     syntax QVarOp ::= QVarSym | "\"" QVarId "\"" [klabel('
        qVarIdQuote)]
51     syntax VarOp ::= VarSym | "\"" VarId "\"" [klabel('
        varIdQuote)]
52     syntax ConOp ::= ConSym | "\"" ConId "\"" [klabel('
        conIdQuote)]
53
54     syntax GConSym ::= ":" | QConSym
55     syntax Vars ::= Var
56                   | Var "," Vars [klabel('varCon)]
57     syntax VarsType ::= Vars "::" Type [klabel('varAssign)]
58     syntax Ops ::= Op
59                   | Op "," Ops [klabel('opCon)]
60     syntax Fixity ::= "infixl" | "infixr" | "infix"
61     syntax Op ::= VarOp | ConOp
62     syntax CQName ::= Var | Con | QVar
63
64     syntax QOp ::= QVarOp | QConOp
65
66     syntax ModuleName ::= "module" ModId [klabel('moduleName)]
67
68     syntax Module ::= ModuleName "where" Body [klabel('
        module)]
69                   | ModuleName Exports "where" Body [klabel('
        moduleExp)]
70                   | Body [klabel('
        moduleBody)]
71
72     syntax Body ::= "{" ImpDecls ";" TopDecls "}" [klabel('
        bodyimpandtop)]
73                   | "{" ImpDecls "}" [klabel('bodyimpdecls)]
74                   | "{" TopDecls "}" [klabel('bodytopdecls)]
75
76     syntax ImpDecls ::= List{ImpDecl, ";"} [klabel('impDecls)]
```



```
77     syntax Exports ::= "(" ExportList OptComma ")"
78     syntax ExportList ::= List{Export, ","}
79
80     syntax Export ::= QVar
81                     | QTyCon OptCQList
82                     | ModuleName
83
84     //optional cname list
85     syntax OptCQList ::= "(.)"
86                       | "(" CQList ")" [klabel('cqListBracket
87                                     //Liyi: a check needs to place in
88                                     preprocessing to check
89                                     //if the CQList is a cname list or a
90                                     qvar list.
91                                     | "" [onlyLabel, klabel('
92                                     emptyOptCNameList)]
93     syntax CQList ::= List{CQName, ","}
94
95     syntax ImpDecl ::= "import" OptQualified ModId OptAsModId
96                     OptImpSpec [klabel('impDecl)]
97                     | "" [onlyLabel, klabel('emptyImpDecl)]
98     syntax OptQualified ::= "qualified"
99                     | "" [onlyLabel, klabel('
100                     emptyQualified)]
101
102     syntax OptAsModId ::= "as" ModId
103                     | "" [onlyLabel, klabel('
104                     emptyOptAsModId)]
105
106     syntax OptImpSpec ::= ImpSpec
107                     | "" [onlyLabel, klabel('
108                     emptyOptImpSpec)]
109
110     syntax ImpSpecKey ::= "(" ImportList OptComma ")"
111     syntax ImpSpec ::= ImpSpecKey
112                     | "hiding" ImpSpecKey
113
114     syntax ImportList ::= List{Import, ","}
115
116     syntax Import ::= Var
117                     | TyCon CQList
118     syntax TopDecls ::= List{TopDecl, ";"} [klabel('topdeclslist
119                                     )]
```

```
111
112   syntax TopDecl ::= Decl [klabel('topdecldecl)]
113                       > "type" SimpleType "=" Type [klabel('type)
114                       ]
115                       | "data" OptContext SimpleType OptConstrs
116                           OptDeriving [klabel('data)]
117                       | "newtype" OptContext SimpleType "="
118                           NewConstr OptDeriving [klabel('newtype)]
119                       | "class" OptContext ConId TyVar OptCDecls
120                           [klabel('class)]
121                       | "instance" OptContext QTyCon Inst
122                           OptIDecls [klabel('instance)]
123                       | "default" Types [klabel('default)]
124                       | "foreign" FDecl [klabel('foreign)]
125
126   syntax FDecl ::= "import" CallConv CusString Var "::" FType
127                       | "import" CallConv Safety CusString Var "::"
128                           FType
129                       | "export" CallConv Safety CusString Var "::"
130                           FType
131
132   //Liyi: fdecl needs to use special function in preprocessing
133   // to get the actually elements from the impent and expent
134       from the CusString
135   //did string analysis
136
137   syntax Safety ::= "unsafe" | "safe"
138
139   syntax CallConv ::= "ccall" | "stdcall" | "cplusplus" | "jvm
140       " | "dotnet"
141
142   syntax FType ::= FrType
143                       | FaType "->" FType // unsure about this one
144                           syntax is ambiguous UNFINISHED
145
146   syntax FrType ::= FaType
147                       | "()"
148
149   syntax FaType ::= QTyCon ATypeList
150
151   //define declaration.
152   syntax OptDecls ::= "where" Decls | "" [onlyLabel, klabel('
153       emptyOptDecls)]
154
155   syntax Decls ::= "{" DeclsList "}" [klabel('decls)]
156
157   syntax DeclsList ::= List{Decl, ";"} [klabel('declsList)]
```

```
143
144     syntax Decl ::= GenDecl
145                   | FunLhs Rhs [klabel('declFunLhsRhs)]
146                   | Pat Rhs [klabel('declPatRhs)]
147
148     syntax OptCDecls ::= "where" CDecls | "" [onlyLabel, klabel
149                   ('emptyOptCDecls)]
150     syntax CDecls ::= "{" CDeclsList "}"
151     syntax CDeclsList ::= List{CDecl, ";"}
152
153     syntax CDecl ::= GenDecl
154                   | FunLhs Rhs
155                   | Var Rhs
156
157     syntax OptIDecls ::= "where" IDecls | "" [onlyLabel, klabel
158                   ('emptyOptIDecls)]
159     syntax IDecls ::= "{" IDeclsList "}"
160     syntax IDeclsList ::= List{IDecl, ";"} [klabel('ideclslist)]
161
162     syntax IDecl ::= FunLhs Rhs [klabel('cdeclFunLhsRhs)]
163                   | Var Rhs [klabel('cdeclVarRhs)]
164                   | "" [onlyLabel, klabel('emptyIDecl)]
165
166     syntax GenDecl ::= VarsType
167                   | Vars "::" Context "=>" Type [klabel('
168                   genAssignContext)]
169                   | Fixity Ops
170                   | Fixity Integer Ops
171                   | "" [onlyLabel, klabel('emptyGenDecl)]
172
173     //three optional data type for the TopDecl data operator.
174     //deriving data type
175     syntax OptDeriving ::= Deriving | "" [onlyLabel, klabel('
176                   emptyDeriving)]
177     syntax Deriving ::= "deriving" DClass
178                   | "deriving" "(" DClassList ")"
179     syntax DClassList ::= List{DClass, ","}
180     syntax DClass ::= QTyCon
181
182     syntax FunLhs ::= Var APatList [klabel('varApatList)]
183                   | Pat VarOp Pat [klabel('patVarOpPat)]
184                   | "(" FunLhs ")" APatList [klabel('
185                   funlhsApatList)]
```

```

181
182     syntax Rhs ::= "=" Exp OptDecls [klabel('eqExpOptDecls)]
183               | GdRhs OptDecls [klabel('gdRhsOptDecls)]
184
185     syntax GdRhs ::= Guards "=" Exp
186                   | Guards "=" Exp GdRhs
187
188     syntax Guards ::= "|" GuardList
189     syntax GuardList ::= Guard | Guard "," GuardList [klabel('
        guardListCon)]
189
190     syntax Guard ::= Pat "<-" InfixExp
191                   | "let" Decls
192                   | InfixExp
193
194     //definition of exp
195     syntax Exp ::= InfixExp
196                 > InfixExp "::" Type [klabel('expAssign)]
197                 | InfixExp "::" Context "=>" Type [klabel('
        expAssignContext)]
198
199     syntax InfixExp ::= LExp
200                     > "-" InfixExp [klabel('minusInfix)]
201                     > LExp QOp InfixExp
202
203     syntax LExp ::= AExp
204                 > "\" APatList "->" Exp [klabel('lambdaFun)]
205                 | "let" Decls "in" Exp [klabel('letIn)]
206                 | "if" Exp OptSemicolon "then" Exp
207                   OptSemicolon "else" Exp [klabel('ifThenElse
        )]
208                 | "case" Exp "of" "{" Alts "}" [klabel('caseOf
        )]
209                 | "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.

```

1
2     syntax OptSemicolon ::= ";" | "" [onlyLabel, klabel('
        emptySemicolon)]
3
4     syntax OptComma ::= "," | "" [onlyLabel, klabel('
        emptyComma)]

```

```
5      syntax AExp ::= QVar [klabel('aexpQVar)]
6                  | GCon [klabel('aexpGCon)]
7                  | Literal [klabel('aexpLiteral)]
8                  > AExp AExp [left, klabel('funApp)]
9                  > QCon "{" FBindList "]"
10                 | AExp "{" FBindList "]" //aexp cannot be qcon
                                UNFINISHED
11                 //Liyi: first, does not understand the
                        syntax, it is the Qcon {FBindlist}
12                 //or QCon? Second, place a check in
                        preprosssing.
13                 //and also check the Fbindlist here
                        must be at least one argument
14                 > "(" Exp ")" [bracket]
15                 | "(" ExpTuple ")"
16                 | "[" ExpList "]"
17                 | "[" Exp OptExpComma ".." OptExp "]"
18                 | "[" Exp "|" Quals "]"
19                 | "(" InfixExp QOp ")"
20                 | "(" QOp InfixExp ")" //qop cannot be - (
                        minus) UNFINISHED
21                 //Liyi: place a check here to check
                        if QOp is a minus
22
23
24      syntax OptExpComma ::= "," Exp | "" [onlyLabel, klabel('
                        emptyExpComma)]
25      syntax OptExp ::= Exp | "" [onlyLabel, klabel('emptyExp)]
26
27      syntax ExpList ::= Exp | Exp "," ExpList [right]
28      syntax ExpTuple ::= Exp "," Exp [right, klabel('
                        twoExpTuple)]
29                  | Exp "," ExpTuple [right, klabel('
                        expTupleCon)]
30
31      //constr datatypes
32      syntax OptConstrs ::= "=" Constrs [klabel('nonemptyConstrs)
                        ] | "" [onlyLabel, klabel('emptyConstrs)]
33      syntax Constrs ::= Constr [klabel('singleConstr)] |
                        Constr "|" Constrs [klabel('multConstr)]
34      syntax Constr ::= Con OptBangATypes [klabel('constrCon)
                        ] // (arity con = k, k 0) UNFINISHED
35                  | SubConstr ConOp SubConstr
```

```
36             | Con "{" FieldDeclList "}"
37
38   syntax NewConstr  ::= Con AType [klabel('newConstrCon)]
39             | Con "{" Var ":" Type "}"
40
41   syntax SubConstr  ::= BType | "!" AType
42   syntax FieldDeclList ::= List{FieldDecl, ", "}
43   syntax FieldDecl ::= VarsType
44             | Vars ":" "!" AType
45
46
47   syntax OptBangATypes ::= List{OptBangAType, " "} [klabel('
      optBangATypes)]
48   syntax OptBangAType ::= OptBang AType [klabel('optBangAType)
      ]
49   syntax OptBang ::= "!" | "" [onlyLabel, klabel('emptyBang)]
50
51   syntax OptContext ::= Context "=>" | "" [onlyLabel, klabel('
      emptyContext)]
52   syntax Context ::= Class
53             | "(" Classes ")"
54
55   syntax Classes ::= List{Class, ", "}
56
57   syntax SimpleClass ::= QTyCon TyVar [klabel('classCon)]
58
59   syntax Class      ::= SimpleClass
60             | QTyCon "(" TyVar ATypeList ")"
61             //Liyi: a check in preprocessing to
62             //check if the Atype list is
63             //empty
64             //it must have at least one item
65
66   //define type and simple type
67   syntax SimpleType ::= TyCon TyVars [klabel('simpleTypeCon)
      ]
68   syntax Type ::= BType
69             | BType "->" Type [klabel('typeArrow)]
70   syntax BType ::= AType
71             | BType AType [klabel('baTypeCon)]
72
73   syntax ATypeList ::= List{AType, ""} [klabel('atypelist)]
```

```

73     syntax AType ::= GTyCon                                [klabel('
        atypeGTyCon)]
74         | TyVar                                           [klabel('
        atypeTyVar)]
75         | "(" TypeTuple ")"                               [klabel('
        atypeTuple)]
76         | "[" Type "]"                                     [klabel('tyList)
        ]
77         | "(" Type ")"                                     [bracket]
78     syntax TypeTuple ::= Type "," Type                    [right,klabel('
        twoTypeTuple)]
79         | Type "," TypeTuple                             [klabel('
        typeTupleCon)]
80     syntax Types ::= List{Type, ","}
81
82     syntax GConCommas ::= "," | "," GConCommas
83     syntax GConCommon ::= "(" | "[" | "(" GConCommas ")" //was
        incorrect syntax
84     syntax GTyCon ::= QTyCon
85         | GConCommon
86         | "(->)"
87
88     syntax GCon ::= GConCommon
89         | QCon
90
91     //inst definition
92     syntax Inst ::= GTyCon
93         | "(" GTyCon TyVars ")" //TyVars must be
        distinct UNFINISHED
94         | "(" TyVarTuple ")" //TyVars must be
        distinct
95         | "[" TyVar "]" [klabel('tyVarList)]
96         | "(" TyVar "->" TyVar ")" //TyVars must be
        distinct
97     //pat definition
98     syntax Pat ::= LPat QConOp Pat
99         | LPat
100
101     syntax LPat ::= APat
102         | "-" IntFloat [klabel('minusPat)]
103         | GCon APatList [klabel('lpatCon)]//arity
        gcon = k UNFINISHED
104

```

```
105     syntax APatList ::= APat | APat APatList [klabel('apatCon)]
106
107     syntax APat ::= Var [klabel('apatVar)]
108                   | Var "@" APat
109                   | GCon
110                   | QCon "{" FPats "}"
111                   | Literal [klabel('apatLiteral)]
112                   | "_"
113                   | "(" Pat ")" [bracket]
114                   | "(" PatTuple ")"
115                   | "[" PatList "]"
116                   | "~" APat
117
118     syntax PatTuple ::= Pat "," Pat [klabel('twoPatTuple
119                               )]
120                               | Pat "," PatTuple [klabel('patTupleCon
121                               )]
122
123     syntax PatList ::= Pat
124                   | Pat "," PatList [klabel('patListCon)
125                   ]
126
127     //definition of quals
128     syntax FPat ::= QVar "=" Pat
129
130     //definition of quals
131     syntax Quals ::= Qual | Qual "," Quals [klabel('qualCon)]
132
133     syntax Qual ::= Pat "<-" Exp
134                   | "let" Decls
135                   | Exp
136
137     //definition of alts
138     syntax Alts ::= Alt | Alt ";" Alts
139
140     syntax Alt ::= Pat "->" Exp [klabel('altArrow)]
141                   | Pat "->" Exp "where" Decls
142                   | "" [onlyLabel, klabel('emptyAlt)]
143
144     //definition of stmts
145     syntax StmtList ::= StmtList Exp OptSemicolon
146     syntax StmtList ::= List{Stmt, ""}
147     syntax Stmt ::= Exp ";"
148                   | Pat "<-" Exp ";"
```



```
145             | "let" Decls ";"  
146             | ";"  
147  
148     //definition of fbind  
149     syntax FBindList ::= List{FBind, ","}  
150     syntax FBind ::= QVar "=" Exp
```

---

## APPENDIX B

### HASKELL-CONFIGURATION.K

```
1 requires "haskell-syntax.k"
2
3 module HASKELL-CONFIGURATION
4     imports HASKELL-SYNTAX
5
6     syntax KItem ::= "startImportRecursion"
7     syntax KItem ::= callInit(K)
8     //syntax KItem ::= initPreModule(K) [function]
9     //syntax KItem ::= tChecker(K) [function]
10
11     configuration
12         <T>
13             <k> $PGM:ModuleList ~> startImportRecursion </k>
14             <tempModule> .K </tempModule>
15             <tempCode> .K </tempCode>
16             <typeIterator> 1 </typeIterator>
17             <tempAlpha> .K </tempAlpha>
18             <tempAlphaMap> .Map </tempAlphaMap>
19             <tempBeta> .Map </tempBeta>
20             <tempT> .K </tempT>
21             <tempDelta> .Map </tempDelta>
22             <tempAlphaStar> .K </tempAlphaStar>
23             <tempBetaStar> .K </tempBetaStar>
24             <importTree> .List </importTree>
25             <recurImportTree> .List </recurImportTree>
26             <impTreeVMap> .Map </impTreeVMap>
27             <modules> //static information about a module
28                 <module multiplicity="*">
29                     <moduleName> .K </moduleName>
30                     <moduleAlphaStar> .K </moduleAlphaStar>
31                     <moduleBetaStar> .K </moduleBetaStar>
32                     <moduleImpAlphas> .List </moduleImpAlphas>
33                     <moduleImpBetas> .List </moduleImpBetas>
```

```
34          <moduleCompCode> .K </moduleCompCode>
35          <moduleTempCode> .K </moduleTempCode>
36          <imports> .Set </imports>
37          <classes> //static information about a
              module
38              <class multiplicity="*">
39                  <className> .K </className>
40              </class>
41          </classes>
42      </module>
43  </modules>
44  </T>
45
46 endmodule
```

---

## APPENDIX C

### HASKELL-PREPROCESSING.K

```
1 //
2 requires "haskell-syntax.k"
3 requires "haskell-configuration.k"
4
5 module HASKELL-PREPROCESSING
6     imports HASKELL-SYNTAX
7     imports HASKELL-CONFIGURATION
8
9     //USER DEFINED LIST
10    //definition of ElemList
11
12    //syntax KItem ::= ElemList
13    syntax ElemList ::= List{Element, ",", ""} [strict]
14 //    syntax Int ::= lengthOfList(ElemList) [function]
15
16 //    rule lengthOfList(.ElemList) => 0
17 //    rule lengthOfList(val(K:K), L:ElemList) => 1 +Int
18 //    rule lengthOfList(valValue(K:K), L:ElemList) => 1 +Int
19 //    lengthOfList(L)
20
21    syntax Element ::= val(K) [strict]
22    syntax ElementResult ::= valValue(K)
23    syntax Element ::= ElementResult
24    syntax KResult ::= ElementResult
25    rule val(K:KResult) => valValue(K) [structural]
26
27    //form ElemList
28    syntax ElemList ::= formElemList(K) [function]
29
30    //CONVERT ~> TO List
31    //list convert
32    syntax List ::= convertToList(K) [function]
```

```
32 //      rule convertToList(.K) => .List
33 //      rule convertToList(A:KItem ~> B:K) => ListItem(A)
      convertToList(B)
34
35
36      syntax KItem ::= dealWithImports(K,K)
37
38      rule <k> 'modListSingle('module(A:K,, B:K)) =>
      dealWithImports(A,B) ...</k>
39
40      (.Bag =>
41          <module>...    //DOT DOT DOT MEANS OVERWRITE ONLY SOME
      OF THE DEFAULTS
42          <moduleName> A </moduleName>
43          ...</module>
44      )
45
46      rule <k> 'modList('module(A:K,, B:K),, C:K) =>
      dealWithImports(A,B) ~> C ...</k>
47
48      (.Bag =>
49          <module>...    //DOT DOT DOT MEANS OVERWRITE ONLY SOME
      OF THE DEFAULTS
50          <moduleName> A </moduleName>
51          ...</module>
52      )
53
54 //      rule dealWithImports(Mod:K, A:K) => callInit(A)
55
56 //      rule <k> dealWithImports(Mod:K, A:K) => callInit(A) ...</k>
      >
57
58      rule <k> dealWithImports(Mod:K, 'bodyimpandtop(A:K,, B:K))
      => .K ...</k>
59      <importTree> L:List => L importListConvert(Mod, A) </
      importTree>
60      <recurImportTree> L:List => L importListConvert(Mod, A)
      </recurImportTree>
61
62      <moduleName> Mod </moduleName>
63      <imports> S:Set (.Set => SetItem(A)) </imports>
64      <moduleTempCode> OldTemp:K => B </moduleTempCode>
65
```

```
66     rule <k> dealWithImports (Mod:K, 'bodyimpdecls (A:K)) => .K
        ...</k>
67     <importTree> L:List => L importListConvert (Mod, A) </
        importTree>
68     <recurImportTree> L:List => L importListConvert (Mod, A)
        </recurImportTree>
69
70     <moduleName> Mod </moduleName>
71     <imports> S:Set (.Set => SetItem(A)) </imports>
72
73 //     rule <k> dealWithImports (Mod:K, 'bodytopdecls (A:K)) =>
        callInit(A) ...</k>
74     rule <k> dealWithImports (Mod:K, 'bodytopdecls (B:K)) => .K
        ...</k>
75
76     <moduleName> Mod </moduleName>
77     <moduleTempCode> OldTemp:K => B </moduleTempCode>
78
79     //importlist convert
80     syntax List ::= importListConvert (K,K) [function]
81     syntax KItem ::= impObject (K,K)
82
83     rule importListConvert (Name:K, 'impDecl (A:K,, Rest:K)) =>
        importListConvert (Name, A) importListConvert (Name, Rest)
84     rule importListConvert ('moduleName (Name:K), 'impDecl (A:K,,
        Modid:K,, C:K,, D:K)) => ListItem (impObject (Name, Modid))
85     rule importListConvert (Name:K, .ImpDecl) => .List
86
87 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
88
89     /*NEW TODO ALGORITHM
90     1. Construct tree for module inclusion
91     2. Check tree for cycles
92     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
93     (and specify scoping) (desugar the scope so that each type
        specifies the scope) */
94
95     syntax KItem ::= "checkImportCycle"
96     syntax KItem ::= "recurseImportTree"
97
98 /*     rule <k> performNextChecks
```

```

99         => checkUseVars
100         ~> (checkLabelUses
101         ~> (checkBlockAddress(.K)
102         ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS
            -Set TES)
103         ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
104         ~> (checkAllExpInFromInvoke(.K, TES)
105         ~> (checkLandingpad
106         ~> checkLandingDomResumes)))))) ...</k> */
107
108 rule <k> startImportRecursion => checkImportCycle
109         ~> (recurseImportTree)...</
            k>
110
111 syntax KItem ::= cycleCheck(K,Map,List,List) [function] //
            current node, map of all nodes to visited or not, stack,
            graph
112 syntax Map ::= createVisitMap(List,Map) [function] //graph,
            visitmap
113 syntax KItem ::= getUnvisitedNode(K,K, Map) [function] //
            visitmap
114 syntax List ::= getNodeNeighbors(K,List) [function] //
            visitmap
115
116 rule <k> checkImportCycle
117     => cycleCheck(.K,createVisitMap(I, .Map),.List,I)
            ...</k>
118     <importTree> I:List </importTree>
119     <impTreeVMap> .Map => createVisitMap(I, .Map) </
            impTreeVMap>
120
121 syntax KItem ::= "visited"
122 syntax KItem ::= "unvisited"
123 syntax KItem ::= "none"
124
125 rule createVisitMap(ListItem(impObject(A:K,B:K)) Rest:List,
            M:Map)
126     => createVisitMap(Rest, M[A <- unvisited][B <-
            unvisited])
127 rule createVisitMap(.List, M:Map) => M
128
129 rule getUnvisitedNode(.K, .K, .Map) => none
130 rule getUnvisitedNode(.K, .K, (A:K |-> B:K) M:Map)
```

```
131         => getUnvisitedNode(A, B, M)
132 rule getUnvisitedNode(A:KItem, unvisited, M:Map) => A
133 rule getUnvisitedNode(A:KItem, visited, M:Map)
134     => getUnvisitedNode(.K, .K, M)
135
136
137
138 rule getNodeNeighbors(Node:K, .List) => .List
139 rule getNodeNeighbors(.K, Rest:List) => .List
140
141 rule getNodeNeighbors(Node:KItem, ListItem(impObject(Node, B:
142     KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
143     ListItem(B)
144 rule getNodeNeighbors(Node:KItem, ListItem(impObject(A:KItem,
145     B:KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
146     requires Node !=K A
147
148 rule cycleCheck(none, M:Map, .List, L:List) => .K
149 rule cycleCheck(.K, M:Map, .List, I:List) => cycleCheck(
150     getUnvisitedNode(.K, .K, M), M, .List, I)
151 rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:List)
152     => cycleCheck(Node, M, S, I)
153 rule cycleCheck(Node:K, M:Map, S:List, I:List)
154     => cycleCheck(.K, M[Node <- visited],
155         getNodeNeighbors(Node, I) S, I)
156     requires Node !=K .K and Bool Node !=K none
157 rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
158     cycleCheck(Node, M, S, I)
159     requires S !=K .List
160 /*
161 rule cycleCheck(A:K, .K, .K, I:K) => cycleCheck(A,
162     createVisitMap(I, .Map), .List, I)
163
164
165
166 rule cycleCheck(Node:K, M:Map, S:List, I:K) => cycleCheck(.K
167     , M[Node <- visited], getNodeNeighbors(Node, I) S, I)
168
169
170 rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
171     cycleCheck(Node, M, S, I)
```



```
163      //rule cycleCheck(.K, M:Map, .K, ListItem(impObject(A:K,B:K)
          ) Rest:List) => cycleCheck(ListItem(impObject(A:K,B:K))
          Rest:List)
164 */
165
166 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

167
168 //COPY IMPORT GRAPH, NEED SECOND GRAPH FOR RECURSING, ADDITIONAL
    GRAPH FOR SELECTING IMPORTS FOR ALPHA* AND BETA*
169 //DFS for leaf
170 //acquire alpha and beta for leaf
171 //merge alpha and beta with imports to produce alpha* and beta*
172 //perform checks
173 //perform inferencing
174 //insert alpha* and beta* into importing modules
175 //remove all edges pointing to leaf
176
177     syntax KItem ::= "leafDFS"
178     syntax KItem ::= "getAlphaAndBeta"
179     syntax KItem ::= "getAlphaBetaStar"
180     syntax KItem ::= "performIndividualChecks"
181     syntax KItem ::= "performIndividualInferencing"
182     syntax KItem ::= "insertAlphaBetaStar"
183     syntax KItem ::= "removeAllEdges"
184     syntax KItem ::= "seeIfFinished"
185
186     rule <k> recurseImportTree => leafDFS
187                               ~> (getAlphaAndBeta
188                               //~> (getAlphaBetaStar
189                               ~> (
                                   performIndividualInferencing
                                   ))...</k>

190
191 //rule <k> dealWithImports (Mod:K, 'bodytopdecls(A:K)) =>
    callInit(A) ...</k>
192
193 //     rule <k> leaf
194 //           => cycleCheck(.K,createVisitMap(I, .Map),.List,I)
    ...</k>
195 //     <importTree> I:List </importTree>
196
```

```

197 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
198
199   syntax KItem ::= returnLeafDFS(K,List,Map) [function] //
        current node, map of all nodes to visited or not, stack,
        graph
200   syntax KItem ::= innerLeafDFS(K,List) [function]
201   syntax KItem ::= loadModule(K)
202
203   rule <k> leafDFS
204       => returnLeafDFS(.K,I,M) ...</k>
205       <recurImportTree> I:List </recurImportTree>
206       <impTreeVMap> M:Map </impTreeVMap>
207
208   rule returnLeafDFS(.K,ListItem(impObject(Node:KItem,B:KItem)
        ) I:List,M:Map) => returnLeafDFS(B,I,M)
209   rule returnLeafDFS(Node:KItem,I:List,M:Map) => returnLeafDFS
        (innerLeafDFS(Node,I),I,M)
210       requires innerLeafDFS(Node,I) !=K none
211   rule returnLeafDFS(Node:KItem,I:List,M:Map) => loadModule(
        Node)
212       requires innerLeafDFS(Node,I) ==K none
213
214   rule innerLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem
        )) I:List) => B
215   rule innerLeafDFS(Node:KItem,ListItem(impObject(A:KItem,B:
        KItem)) I:List) => innerLeafDFS(Node,I)
216       requires Node !=K A
217   rule innerLeafDFS(Node:KItem,.List) => none
218 //   returnLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem))
        I:List,M:Map) => returnLeafDFS(B,I,M)
219
220
221 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
222
223   //call before Checker Code
224 //   rule <k> callInit(S:K) => initPreModule(S) ...</k>
225 //       <tempModule> A:K => S </tempModule>
226
227   rule <k> loadModule(S:KItem) => .K ...</k>
228       <tempModule> A:K => S </tempModule>
229

```

```
230     rule <k> getAlphaAndBeta => initPreModule(Code) ...</k>
231         <tempModule> Mod:KItem </tempModule>
232
233         <moduleName> 'moduleName (Mod) </moduleName>
234         <moduleTempCode> Code:KItem </moduleTempCode>
235
236
237 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

238     //get alpha and beta
239     syntax KItem ::= Module(K, K)
240     syntax KItem ::= preModule(K,K) //(alpha, T)
241
242     // STEP 1 CONSTRUCT T AND ALPHA
243     // alpha = type
244     // T = newtype and data, temporary data structure
245
246     syntax KItem ::= initPreModule(K) [function]
247     syntax KItem ::= getPreModule(K, K) [function] //(Current
        term, premodule)
248     syntax KItem ::= makeT (K,K,K,K)
249
250     syntax KItem ::= fetchTypes (K,K,K,K)
251
252     syntax List ::= makeInnerT (K,K,K) [function] //LIST
253     syntax List ::= getTypeVars(K) [function] //LIST
254
255     syntax KItem ::= getCon(K) [function]
256     syntax List ::= getArgSorts(K) [function] //LIST
257
258     syntax KItem ::= AList(K)
259     syntax KItem ::= AObject(K,K) //(1st -> 2nd) map without
        idempotency
260     syntax KItem ::= ModPlusType(K,K)
261
262     syntax KItem ::= TList(K) //list of T objects for every new
        type introduced by data and newtype
263     syntax KItem ::= TObject(K,K,K,K) //(module name, type name,
        entire list of poly type vars, list of inner T pieces)
264     syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type constructor,
        poly type vars, argument sorts, entire constr block,
        type name)
265
```

```

266 //      rule initPreModule('module(I:ModuleName,, J:K)) =>
      getPreModule(J,preModule(AList(.List),TList(.List)))
267 //      rule initPreModule('moduleExp(I:ModuleName,, L:K,, J:K))
      => getPreModule(J,preModule(AList(.List),TList(.List)))
268 //      rule initPreModule('moduleBody(J:Body)) => getPreModule(J,
      preModule(AList(.List),TList(.List)))
269
270      rule initPreModule(J:K) => getPreModule(J,preModule(AList(.
      List),TList(.List)))
271
272      rule getPreModule('bodytopdecls(I:K), J:K) => getPreModule(I
      ,J)
273      rule getPreModule('topdeclslist('type(A:K,, B:K),, Rest:K),J
      :K) => fetchTypes(A,B,Rest,J) //constructalpha
274
275
276      rule getPreModule('topdeclslist('data(A:K,, B:K,, C:K,, D:K)
      ,, Rest:K),J:K) => makeT(B,C,Rest,J)
277      rule getPreModule('topdeclslist('newtype(A:K,, B:K,, C:K,, D
      :K),, Rest:K),J:K) => makeT(B,C,Rest,J)
278
279
280      rule getPreModule('topdeclslist('topdecldecl(A:K),, Rest:K),
      J:K) => getPreModule(Rest,J)
281      rule getPreModule('topdeclslist('class(A:K,, B:K,, C:K,, D:K
      ),, Rest:K),J:K) => getPreModule(Rest,J)
282      rule getPreModule('topdeclslist('instance(A:K,, B:K,, C:K,,
      D:K),, Rest:K),J:K) => getPreModule(Rest,J)
283      rule getPreModule('topdeclslist('default(A:K,, B:K,, C:K,, D
      :K),, Rest:K),J:K) => getPreModule(Rest,J)
284      rule getPreModule('topdeclslist('foreign(A:K,, B:K,, C:K,, D
      :K),, Rest:K),J:K) => getPreModule(Rest,J)
285      rule getPreModule(.TopDecls,J:K) => J
286
287      //rule getPreModule('module(I:ModuleName,L:K, J:K)) =>
      preModule(J)
288
289      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>
290      <tempModule> ModName:Kitem </tempModule>
291

```

```

292     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(
      TObjct(ModName,I,H,makeInnerT(I,H,D))) ListInside)))
      ...</k>
293     <tempModule> ModName:KItem </tempModule>
294
295     rule makeInnerT(A:K,B:K,'nonemptyConstrs(C:K)) => makeInnerT
      (A,B,C)
296     rule makeInnerT(A:K,B:K,'singleConstr(C:K)) => ListItem(
      InnerTPiece(getCon(C),getTypeVars(C),getArgSorts(C),C,A))
297     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)
298
299     rule getTypeVars('constrCon(A:K,, B:K)) => getTypeVars(B)
300     rule getTypeVars('optBangATypes(A:K,, Rest:K)) =>
      getTypeVars(A) getTypeVars(Rest)
301     rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K))
      => getTypeVars(Rest)
302     rule getTypeVars('atypeGTyCon(A:K)) => .List
303     rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
304     rule getTypeVars(.OptBangATypes) => .List
305
306     //rule getCon('emptyConstrs()) => .K
307     //rule getCon('nonemptyConstrs(A:K)) => getCon(A)
308     rule getCon('constrCon(A:K,, B:K)) => A
309
310     //rule getArgSorts('constrCon(A:K,, B:K)) => B
311     rule getArgSorts('constrCon(A:K,, B:K)) => getArgSorts(B)
312     rule getArgSorts('optBangATypes(A:K,, Rest:K)) =>
      getArgSorts(A) getArgSorts(Rest)
313     rule getArgSorts('optBangAType('emptyBang(.KList),, Rest:K))
      => getArgSorts(Rest)
314     rule getArgSorts('atypeGTyCon(A:K)) => ListItem(A)
315     rule getArgSorts('atypeTyVar(A:K)) => .List
316     rule getArgSorts(.OptBangATypes) => .List
317
318     //////////////////////////////////////
319
320     rule <k> preModule(A:K,T:K) => startTTransform ...</k>
321     <tempAlpha> OldAlpha:K => A </tempAlpha>

```

```

322         <tempT> OldT:K => T </tempT>
323
324 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
325
326 // STEP 2 PERFORM CHECKS
327
328 syntax KItem ::= "error"
329
330 syntax KItem ::= "startChecks"
331 syntax KItem ::= "checkNoSameKey"
332     //Keys of alpha and keys of T should be unique
333 syntax KItem ::= "checkTypeConsDontCollide"
334     //Make sure typeconstructors do not collide in T
335 syntax KItem ::= "makeAlphaMap"
336     //make map for alpha
337 syntax KItem ::= "checkAlphaNoLoops"
338     //alpha check for no loops
339     //check alpha to make sure that everything points to a T
340 syntax KItem ::= "checkArgSortsAreTargets"
341     //Make sure argument sorts [U] [W,V] are in the set
342     of keys of alpha and targets of T, (keys of T)
343 syntax KItem ::= "checkParUsed"
344 //NEED TO CHECK all the polymorphic parameters from right appear
345 on left. RIGHT SIDE ONLY
346 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
347
348 //      rule <k> preModule(A:K,T:K) => startChecks ...</k>
349 //      <tempAlpha> OldAlpha:K => A </tempAlpha>
350 //      <tempT> OldT:K => T </tempT>
351
352 /*      rule <k> performNextChecks
353         => checkUseVars
354             ~> (checkLabelUses
355                 ~> (checkBlockAddress(.K)
356                     ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS
357                         -Set TES)
358                         ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
359                             ~> (checkAllExpInFromInvoke(.K, TES)
360                                 ~> (checkLandingpad
361                                     ~> checkLandingDomResumes)))))) ...</k> */

```

```
361     rule <k> startChecks
362         => checkNoSameKey
363         ~> (checkTypeConsDontCollide
364             ~> (makeAlphaMap
365                 ~> (checkAlphaNoLoops
366                     ~> (checkArgSortsAreTargets
367                         ~> (checkParUsed)))) ...</k>
368
369     rule <k> checkTypeConsDontCollide
370         => tyConCollCheck(T,.List,.Set) ...</k>
371         <tempT> T:K </tempT>
372
373     //syntax KItem ::= tChecker(K) [function]
374     syntax KItem ::= tyConCollCheck(K,K,K) [function] //(TList,
375         List of Tycons,Set of Tycons)
376     syntax KItem ::= lengthCheck(K,K) [function]
377     //syntax KItem ::= tyConCollCheck(K,K,K) [function]
378     //syntax K ::= innerCollCheck(K) [function]
379     //syntax K ::= tyConCollCheckPasser(K, K) [function]
380
380     //rule tChecker(preModule(Alpha:Map,T:K,Mod:K)) =>
381         tyConCollCheck(innerCollCheck(T),preModule(Alpha,T,Mod))
382
382     //rule tyConCollCheck(.K,preModule(Alpha:Map,H:K,Mod:K)) =>
383         tyConCollCheck(innerCollCheck(H),preModule(Alpha,H,Mod))
384
384     rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:
385         K,ListItem(InnerTPiece(Ty:K,E:K,F:K,H:K,G:K)) Inners:List
386         )) Rest:List),J:Set,D:Set) =>
387         tyConCollCheck(TList(ListItem(TObject(
388             ModName,A,B,Inners)) Rest),ListItem(Ty) J
389             , SetItem(Ty) D)
390
390     rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:
391         K,.List)) Rest:List),J:Set,D:Set) =>
392         tyConCollCheck(TList(Rest),J,D)
393
393     rule tyConCollCheck(TList(.List),J:Set,D:Set) =>
394         lengthCheck(size(J),size(D))
395
395     rule lengthCheck(A:Int, B:Int) => .K
396         requires A ==Int B
397
397     rule lengthCheck(A:Int, B:Int) => error
398         requires A /=Int B
```

```

396
397 //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)
398 syntax KItem ::= keyCheck(K,K,K,K) [function] //(Alpha, T,
      List of names, Set of names)
399
400 rule <k> checkNoSameKey
401     => keyCheck(A, T, .Set, .List) ...</k>
402     <tempAlpha> A:K </tempAlpha>
403     <tempT> T:K </tempT>
404 //rule <k> checkAlphaNoSameKey
405 //     => akeyCheck(.K, .Set) ...</k>
406
407 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)
408 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)
409 rule keyCheck(AList(.List), TList(.List), D:Set, G:List) =>
      lengthCheck(size(G),size(D))
410
411
412 syntax KItem ::= makeAlphaM(K,K) [function] //(Alpha,
      AlphaMap)
413 syntax KItem ::= tAlphaMap(K) //(AlphaMap) temp alphamap
414
415 rule <k> makeAlphaMap
416     => makeAlphaM(A, .Map) ...</k>
417     <tempAlpha> A:K </tempAlpha>
418
419 rule makeAlphaM(AList(ListItem(AObject(A:K,B:K)) C:List), M:
      Map) => makeAlphaM(AList(C), M[A <- B])
420 rule makeAlphaM(AList(.List), M:Map) => tAlphaMap(M)
421
422 rule <k> tAlphaMap(M:K) => .K ...</k>
423     <tempAlphaMap> OldAlphaMap:K => M </tempAlphaMap>
424
425 // syntax KItem ::= tkeyCheck(K,K,K) [function] //(T,List of
      T,Set of T)
426
427 // rule <k> checkTNoSameKey

```



```

428 //          => tkeyCheck(T, .Set, T) ...</k>
429 //          <tempT> T:K </tempT>
430
431 //      rule tkeyCheck(TList(ListItem(TObject(A:K,B:K,C:K)) Rest:
List), D:Set, G:K) => tkeyCheck(TList(Rest), SetItem(A) D, G)
432 //      rule tkeyCheck(TList(.List), D:Set, TList(G:List)) =>
lengthCheck(size(G),size(D))
433
434      syntax KItem ::= aloopCheck(K,K,K,K,K,K,K) [function] //(
      Alpha,List of Alpha,Set of Alpha,CurrNode,lengthcheck,T,
      BigSet)
435
436      rule <k> checkAlphaNoLoops
437          => aloopCheck(A,.List,.Set,.K,.K,T,.Set) ...</k>
438          <tempAlphaMap> A:K </tempAlphaMap>
439          <tempT> T:K </tempT>
440
441      //aloopCheck set and list to check cycles
442      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)
443      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)
444
445      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
446          requires (notBool H in keys(Alpha)) andBool (H in
      typeSet(T, .Set) orBool H in S)
447
448      rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K,T:K,S:
Set) => error //terminal alpha rename is not in T ERROR
449          requires (notBool H in keys(Alpha)) andBool (notBool (H
      in typeSet(T, .Set) orBool H in S))
450
451
452      syntax Set ::= typeSet(K,K) [function] //(K, KSet)
453      rule typeSet(TList(ListItem(TObject(ModName:K, A:K,B:K,C:K))
      Rest:List), D:Set) => typeSet(TList(Rest), SetItem(A) D)
454      rule typeSet(TList(.List), D:Set) => D
455

```

```
456 //      rule aloopCheck(Alpha:Map, D:List, G:Set, H:KItem, .K) =>
      keys(Alpha) ~> H
457 //          requires notBool H in keys(Alpha)
458
459      rule aloopCheck(.Map, .List, .Set, .K, .K,T:K, S:Set) => .K
460 //      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)
461
462
463 //      syntax KItem ::= TList(K) //list of T objects for every
      new type introduced by data and newtype
464 //      syntax KItem ::= TObject(K,K,K) //(type name, entire list
      of poly type vars, list of inner T pieces)
465 //      syntax KItem ::= InnerTPiece(K,K,K,K,K) //(type
      constructor, poly type vars, argument sorts, entire constr
      block, type name)
466
467 //Make sure argument sorts [U] [W,V] are in the set of keys of
      alpha and targets of T, (keys of T)
468
469      syntax KItem ::= argSortCheck(K,K,K) [function] //(T,
      AlphaMap)
470
471      rule <k> checkArgSortsAreTargets
472          => argSortCheck(T,A,typeSet(T,.Set)) ...</k>
473          <tempAlphaMap> A:K </tempAlphaMap>
474          <tempT> T:K </tempT>
475
476      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)
477          requires ((Arg in keys(AlphaMap)) orBool (Arg in Tset))
478
479      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
480          requires (notBool ((Arg in keys(AlphaMap)) orBool (Arg
      in Tset)))
```

```

481
482     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)
483
484     rule argSortCheck (TList (ListItem (TObject (ModName:K, A:K, B:K, .
        List)) TListRest:List), AlphaMap:Map, Tset:Set) =>
        argSortCheck (TList (TListRest), AlphaMap, Tset)
485
486     rule argSortCheck (TList (.List), AlphaMap:Map, Tset:Set) => .K
487
488 //NEED TO CHECK all the polymorphic parameters from right appear
        on left. RIGHT SIDE ONLY
489 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
490
491     syntax KItem ::= parCheck (K, K) [function] // (T, AlphaMap)
492     syntax KItem ::= makeTyVarList (K, K, K) [function] // (TyVars,
        NewList)
493     syntax KItem ::= lengthRet (K, K, K) [function]
494
495     rule <k> checkParUsed
496         => parCheck (T, .K) ...</k>
497     <tempT> T:K </tempT>
498
499 //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)
500     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))
501
502     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) =>
503         parCheck (TList (ListItem (TObject (ModName, A, B, ListItem (
        InnerTPiece (C, ParRest, D, E, F)) InnerRest)) Rest),
        NewSet)
504         requires Par in NewSet
505

```

```

506     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
507         requires notBool (Par in NewSet)
508
509     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) =>
510         parCheck (TList (ListItem (TObject (ModName:A, B, InnerRest))
      Rest), NewSet)
511
512     rule parCheck (TList (ListItem (TObject (ModName:K, A:K, B:K, .List
      )) Rest:List), NewSet:Set) =>
513         parCheck (TList (Rest), NewSet)
514
515     rule parCheck (TList (.List), NewSet:Set) => .K
516
517     rule makeTyVarList ('typeVars (A:K, , Rest:K), NewList:List,
      NewSet:Set) => makeTyVarList (Rest, ListItem (A) NewList,
      SetItem (A) NewSet)
518
519     rule makeTyVarList (.TyVars, NewList:List, NewSet:Set) =>
      lengthRet (size (NewList), size (NewSet), NewSet)
520
521     rule lengthRet (A:Int, B:Int, C:K) => C
522         requires A ==Int B
523
524     rule lengthRet (A:Int, B:Int, C:K) => error
525         requires A /=Int B
526
527     //rule argSortCheck (TList (ListItem (TObject (A:K, B:K, C:K)
528
529     //////////////////////////////////////
530
531     // STEP 3 Transform T into beta
532
533     syntax KItem ::= "startTTransform"
534     syntax KItem ::= "constructDelta"
535     syntax KItem ::= "constructBeta"
536
537     rule <k> startTTransform

```

```
538         => constructDelta
539         ~> (constructBeta) ...</k>
540
541     rule <k> constructDelta
542         => makeDelta(T, .Map) ...</k>
543     <tempT> T:K </tempT>
544
545     syntax KItem ::= makeDelta(K, Map) [function] //(T, Delta)
546     syntax KItem ::= newDelta(Map) //Delta
547     syntax KItem ::= newBeta(Map) //beta
548     syntax List ::= retPolyList(K, List) [function] //(T, Delta)
549
550     rule makeDelta(TList(ListItem(TObject(ModName:K, A:K, Polys:K,
551         C:K)) Rest:List), M:Map) =>
552         makeDelta(TList(Rest), M[ModPlusType(ModName, A) <- size(
553             retPolyList(Polys, .List))])
554     rule makeDelta(TList(.List), M:Map) => newDelta(M)
555
556     rule retPolyList('typeVars(A:K, , Rest:K), NewList:List) =>
557         retPolyList(Rest, ListItem(A) NewList)
558     rule retPolyList(.TyVars, L:List) => L
559
560     rule <k> newDelta(M:Map)
561         => .K ...</k>
562     <tempDelta> OldDelta:K => M </tempDelta>
563
564     rule <k> constructBeta
565         => makeBeta(T, .Map) ...</k>
566     <tempT> T:K </tempT>
567
568     syntax KItem ::= makeBeta(K, Map) [function] //(T, Beta, Delta)
569
570     rule makeBeta(TList(ListItem(TObject(ModName:K, A:K, B:K,
571         ListItem(InnerTPiece(Con:K, H:K, D:K, E:K, F:K)) InnerRest:
572         List)) Rest:List), Beta:Map) =>
573         makeBeta(TList(ListItem(TObject(ModName, A, B, InnerRest))
574             Rest), Beta[ModPlusType(ModName, Con) <- betaParser(E
575                 , B, A)])
576     rule makeBeta(TList(ListItem(TObject(ModName:K, A:K, B:K, .List
577         )) Rest:List), Beta:Map) =>
578         makeBeta(TList(Rest), Beta)
579     rule makeBeta(TList(.List), Beta:Map) =>
580         newBeta(Beta)
```

```
573 //      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) =>
574 //      makeBeta(TList(ListItem(TObject(ModName,A,B,InnerRest
      )) Rest),Beta)
575
576      syntax KItem ::= betaParser(K,K,K) [function] //(Tree Piece,
      NewSyntax,Parameters,Constr)
577      syntax Set ::= getTyVarsRHS(K,List) [function]
578
579      syntax KItem ::= forAll(Set,K)
580      syntax KItem ::= funtype(K,K)
581
582      syntax Set ::= listToSet(List, Set) [function]
583
584      rule listToSet(ListItem(A:KItem) L:List, S:Set) => listToSet
      (L, SetItem(A) S)
585      rule listToSet(.List, S:Set) => S
586
587
588 //if optbangATypes, need to see if first variable is a typecon
589 //if its a typecon then need to go into Delta and see the amount
      of parameters it has
590 //then count the number of optbangATypes after the typecon
591      rule betaParser('constrCon(A:K,, B:K), Par:K, Con:K) =>
      forAll(getTyVarsRHS(B,.List), betaParser(B, Par, Con))
592      rule betaParser('optBangATypes('optBangAType('emptyBang(.
      KList),, 'atypeTyVar(Tyv:K)),, Rest:K), Par:K, Con:K) =>
      funtype(Tyv, betaParser(Rest, Par, Con))
593      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))
594      rule betaParser('optBangATypes('optBangAType('emptyBang(.
      KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K, Con:K) =>
      funtype(Tyc, betaParser(Rest, Par, Con))
595      rule betaParser(.OptBangATypes, Par:K, Con:K) => '
      simpleTypeCon(Con,, Par)
596 //      rule betaParser('optBangATypes('optBangAType('emptyBang(.
      KList),, 'atypeGTyCon(Tyc:K)),, Rest:KItem)) => getTypeVars(A
      ) getTypeVars(Rest)
597 //      rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K
      )) => getTypeVars(Rest)
```

```
598 //      rule getTypeVars(' atypeGTyCon(A:K)) => .List
599 //      rule getTypeVars(' atypeTyVar(A:K)) => ListItem(A)
600 //      rule getTypeVars(.OptBangATypes) => .List
601
602      rule getTyVarsRHS(.OptBangATypes, Tylist:List) => listToSet(
        Tylist, .Set)
603
604      rule <k> newBeta(M:Map)
605          => .K ...</k>
606          <tempBeta> OldBeta:K => M </tempBeta>
607
608 //////////////////////////////////////
609
610 //      syntax KItem ::= "insertAlphaBetaStar"
611
612      syntax KItem ::= insertABRec(K, List)
613      syntax KItem ::= insertAB(K)
614
615      rule <k> insertAlphaBetaStar => insertABRec(Mod, Imp) ...</k>
        >
616          <tempModule> Mod:KItem </tempModule>
617          <importTree> Imp:List </importTree>
618
619      rule <k> insertABRec(Node:KItem, ListItem(impObject(B:KItem,
        Node)) I:List) => insertAB(B) ~> insertABRec(Node, I)
        ...</k>
620
621      rule <k> insertABRec(Node:KItem, ListItem(impObject(B:KItem,
        C:KItem)) I:List) => insertABRec(Node, I) ...</k>
622          requires Node !=K C
623
624      rule <k> insertAB(B) => .K ...</k>
625
626          <tempAlphaStar> Alph:KItem </tempAlphaStar>
627          <tempBetaStar> Bet:KItem </tempBetaStar>
628
629          <moduleName> 'moduleName(B) </moduleName>
630          <moduleImpAlphas> ImpAlphas:List => ListItem(Alph)
        ImpAlphas </moduleImpAlphas>
631          <moduleImpBetas> ImpBetas:List => ListItem(Bet)
        ImpBetas </moduleImpBetas>
632
```

633

634 `endmodule`

---



## APPENDIX D

### HASKELL-TYPE-INFERENCING.K

```
1  requires "haskell-syntax.k"
2  requires "haskell-configuration.k"
3  requires "haskell-preprocessing.k"
4
5  module HASKELL-TYPE-INFERENCING
6      imports HASKELL-SYNTAX
7      imports HASKELL-CONFIGURATION
8      imports HASKELL-PREPROCESSING
9
10     syntax KItem ::= "Bool" //Boolean
11
12     // STEP 4 Type Inferencing
13     syntax KItem ::= inferenceShell(K) [function]//Input,
        AlphaMap, Beta, Delta, Gamma
14     //syntax KItem ::= typeInferenceFun(K,Map,Map,Map,Map,K,K) [
        function]//Input, Alpha, Beta, Delta, Gamma
15     //syntax KItem ::= typeInferenceFun(Map,K,K) //Gamma,
        Expression, Guessed Type
16     syntax Map ::= genGamma(K,Map,K) [function] //Apatlist,
        Gamma Type
17     syntax KItem ::= genLambda(K,K) [function]
18     syntax KItem ::= guessType(Int)
19 //     syntax KItem ::= lambdaReturn(K,K,K)
20     syntax KItem ::= freshInstance(K, Int) [function]
21     syntax Int ::= paramSize(K) [function]
22
23
24     syntax KItem ::= mapBag(Map)
25     syntax KResult ::= mapBagResult(Map)
26
27     syntax Map ::= gammaSub(Map,Map,Map) [function]//
        substitution, gamma
28
```

```
29     rule <k> performIndividualInferencing => inferenceShell(Code
30         ) ...</k>
31         <tempModule> Mod:KItem </tempModule>
32         <moduleName> 'moduleName(Mod) </moduleName>
33         <moduleTempCode> Code:KItem </moduleTempCode>
34
35     rule inferenceShell('topdeclslist('type(A:K,, B:K),, Rest:K)
36         ) =>
37         inferenceShell(Rest) //constructalpha
38     rule inferenceShell('topdeclslist('data(A:K,, B:K,, C:K,, D:
39         K),, Rest:K)) =>
40         inferenceShell(Rest)
41     rule inferenceShell('topdeclslist('newtype(A:K,, B:K,, C:K,,
42         D:K),, Rest:K)) =>
43         inferenceShell(Rest)
44     rule inferenceShell('topdeclslist('class(A:K,, B:K,, C:K,, D
45         :K),, Rest:K)) =>
46         inferenceShell(Rest)
47     rule inferenceShell('topdeclslist('instance(A:K,, B:K,, C:K
48         ,, D:K),, Rest:K)) =>
49         inferenceShell(Rest)
50     rule inferenceShell('topdeclslist('default(A:K,, B:K,, C:K,,
51         D:K),, Rest:K)) =>
52         inferenceShell(Rest)
53     rule inferenceShell('topdeclslist('foreign(A:K,, B:K,, C:K,,
54         D:K),, Rest:K)) =>
55         inferenceShell(Rest)
56     rule inferenceShell('topdeclslist('topdecldecl(A:K),, Rest:K
57         )) =>
58         typeInferenceFun(.ElemList, .Map,A,guessType(0)) ~>
59         inferenceShell(Rest)
```

```
52
53
54     rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
55         declFunLhsRhs(Fn:K,, Lhsrhs:K), Guess:K) =>
56         typeInferenceFun(.ElemList, Gamma, Lhsrhs, Guess) ...</
57         k>
58     rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
59         eqExpOptDecls(Ex:K,, Optdecls:K), Guess:K) =>
60         typeInferenceFun(.ElemList, Gamma, Ex, Guess) ...</k>
```

```
59 //T-App
60 //rule typeInferenceFun('aexpQVar(Var:K), Alpha:Map, Beta:
    Map, Delta:Map, (Var |-> Sigma:K) Gamma:Map,.K,.K) =>
    Sigma
61 //Gamma Proves x:phi(tau) if Gamma(x) = \forall alpha_1,
    ..., alpha_n . tau
62 //where phi replaces all occurrences of alpha_1, ...,
    alpha_n by monotypes tau_1, ..., tau_n
63
64 rule <k> typeInferenceFun(.ElemList, (Var |-> Type:K) Gamma:
    Map, 'aexpQVar(Var:K), Guess:KItem)
65     => mapBagResult(unifun(ListItem(unipair(Guess,
        freshInstance(Type, TypeIt)))) ...</k> //Variable
        rule
66     <typeIterator> TypeIt:Int => TypeIt +Int paramSize(Type
        ) </typeIterator>
67
68 //rule typeInferenceFun('aexpGCon(Gcon:K), Alpha:Map, (Gcon
    |-> Sigma:K) Beta:Map, Delta:Map, Gamma:Map,.K,.K) =>
    Sigma //T-App
69 //rule typeInferenceFun('aexpGCon(Gcon:K), Alpha:Map, Lol:
    Map, Delta:Map, Gamma:Map,.K,.K) => Sigma //T-App
70 //     <tempBeta> (Gcon |-> Sigma:K) Beta:Map </tempBeta>
71
72 rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'aexpGCon('
    conTyCon(Mid:K,, Gcon:K)), Guess:KItem)
73     => mapBagResult(unifun(ListItem(unipair(Guess,
        freshInstance(Type, TypeIt)))) ...</k> //Constant
        rule
74     <tempBeta> (ModPlusType(Mid,Gcon) |-> Type:K) Beta:Map
        </tempBeta>
75     <typeIterator> TypeIt:Int => TypeIt +Int paramSize(Type
        ) </typeIterator>
76
77 //lambda rule
78 //     rule <k> typeInferenceFun(Gamma:Map, 'lambdaFun(Apatlist:K
    ,, Ex:K), Guess:KItem)
79 //         => typeInferenceFun(genGamma(Apatlist,Gamma,
    guessType(TypeIt)), genLambda(Apatlist,Ex), guessType(TypeIt
    +Int 1))
80 //         ~> lambdaReturn(Guess,guessType(TypeIt),guessType(
    TypeIt +Int 1)) ...</k>
```

```
81 //      <typeIterator> TypeIt:Int => TypeIt +Int 2 </
    typeIterator>
82
83 //      rule <k> Sigma:Map ~> lambdaReturn(Tau:K, Tauone:K, Tautwo
    :K)
84 //      => compose (uniFun (ListItem (uniPair (typeSub (Sigma,Tau
    ),typeSub (Sigma,funtype (Tauone,Tautwo))))),Sigma) ...</k>
85
86 syntax KItem ::= typeInferenceFun (ElemList, Map, K, K) [
    strict(1)]
87 syntax KItem ::= typeInferenceFunLambda (ElemList, K, K, K) [
    strict(1)]
88 /* automatically generated by the strict(1) in typeInferenceFun
    or typeInferenceFunAux
89 rule typeInferenceFunAux (Es:ElemList, C:K, A:K, B:K) => Es ~>
    typeInferenceFun (HOLE, C, A, B)
90     requires notBool isKResult (Es)
91 rule Es:KResult ~> typeInferenceFunAux (HOLE, C:K,A:K, B:K) =>
    typeInferenceFun (Es, C, A, B)
92 */
93
94 //lambda rule
95 rule <k> typeInferenceFun (.ElemList, Gamma:Map, 'lambdaFun (
    Apatlist:K,, Ex:K), Guess:KItem)
96     => 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>
97     <typeIterator> TypeIt:Int => TypeIt +Int 2 </
    typeIterator>
98
99 rule <k> typeInferenceFunLambda (valValue (mapBagResult (Sigma:
    Map)), .ElemList, Tau:K, Tauone:K, Tautwo:K)
100     => mapBagResult (compose (uniFun (ListItem (uniPair (typeSub
    (Sigma,Tau),typeSub (Sigma,funtype (Tauone,Tautwo))))
    ,Sigma)) ...</k>
101
102 //rule <k> substi (S:Map) ~> lambdaReturn (Tau:K, Tauone:K,
    Tautwo:K)
103 //      => S [Tauone] ...</k>
104
105
```

```
106 //syntax KItem ::= appliReturn(Map, K, K, Map)
107 //syntax KItem ::= typeChildSub(Map, K) [function]
108
109 syntax KItem ::= typeInferenceFunAppli(ElemList, Map, K, K,
    Map) [strict(1)]
110
111 //application rule
112 rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'funApp(Eone
    :K,, Etwo:K), Guess:KItem)
113     => typeInferenceFunAppli(val(typeInferenceFun(.
        ElemList, Gamma, Eone, funtype(guessType(TypeIt),
        Guess))), .ElemList, Gamma, Etwo, guessType(TypeIt)
        , .Map) ...</k>
114     <typeIterator> TypeIt:Int => TypeIt +Int 1 </
        typeIterator>
115
116 rule <k> typeInferenceFunAppli(valValue(mapBagResult(
    Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem,
    guessType(TypeIt:Int), .Map)
117     => typeInferenceFunAppli(val(typeInferenceFun(.
        ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo,
        typeSub(Sigmaone, guessType(TypeIt))), .ElemList,
        .Map, .K, .K, Sigmaone) ...</k>
118
119 rule <k> typeInferenceFunAppli(valValue(mapBagResult(
    Sigmatwo:Map)), .ElemList, .Map, .K, .K, Sigmaone:Map)
120     => mapBagResult(compose(Sigmatwo, Sigmaone)) ...</k>
121
122 // rule <k> Sigmaone:Map ~> appliReturn(Gamma:Map, Etwo:KItem
    , guessType(TypeIt:Int), .Map)
123 //     => typeInferenceFun(gammaSub(Sigmaone, Gamma, .Map),
    Etwo, typeSub(Sigmaone, guessType(TypeIt)))
124 //     ~> appliReturn(.Map, .K, .K, Sigmaone) ...</k>
125
126 // rule <k> Sigmatwo:Map ~> appliReturn(.Map, .K, .K,
    Sigmaone:Map)
127 //     => compose(Sigmatwo, Sigmaone) ...</k>
128
129 syntax KItem ::= typeInferenceFunIfThen(ElemList, Map, K, K,
    K, Map, Map) [strict(1)]
130
131 //if_then_else rule
```

```
132     rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'ifThenElse(
        Eone:K,, Optsem:K,, Etwo:K,, Optsemtwo:K,, Ethree:K),
        Guess:KItem)
133     => typeInferenceFunIfThen(val(typeInferenceFun(.
        ElemList, Gamma, Eone, Bool)), .ElemList, Gamma,
        Etwo, Ethree, Guess, .Map, .Map) ...</k>
134
135     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
        Sigmaone:Map)), .ElemList, Gamma:Map, Etwo:KItem, Ethree:
        KItem, Guess:KItem, .Map, .Map)
136     => typeInferenceFunIfThen(val(typeInferenceFun(.
        ElemList, gammaSub(Sigmaone, Gamma, .Map), Etwo,
        typeSub(Sigmaone, Guess)), .ElemList, Gamma, .K,
        Ethree, Guess, Sigmaone, .Map) ...</k>
137
138     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
        Sigmatwo:Map)), .ElemList, Gamma:Map, .K, Ethree:KItem,
        Guess:KItem, Sigmaone:Map, .Map)
139     => typeInferenceFunIfThen(val(typeInferenceFun(.
        ElemList, gammaSub(compose(Sigmatwo, Sigmaone),
        Gamma, .Map), Ethree, typeSub(compose(Sigmatwo,
        Sigmaone), Guess)), .ElemList, .Map, .K, .K, .K,
        Sigmaone, Sigmatwo) ...</k>
140
141     rule <k> typeInferenceFunIfThen(valValue(mapBagResult(
        Sigmathree:Map)), .ElemList, .Map, .K, .K, .K, Sigmaone:
        Map, Sigmatwo:Map)
142     => mapBagResult(compose(compose(Sigmathree, Sigmatwo),
        Sigmaone)) ...</k>
143
144     syntax KItem ::= typeInferenceFunLetIn(ElemList, Map, Map, K
        , K, K, Int, Int, Map, Map) [strict(1)]
145     syntax KItem ::= grabLetDeclName(K, Int) [function]
146     syntax KItem ::= grabLetDeclExp(K, Int) [function]
147     syntax KItem ::= mapLookup(Map, K) [function]
148     syntax Map ::= makeDeclMap(K, Int, Map) [function]
149     syntax Map ::= applyGEN(Map, Map, Map, Map) [function]
150
151     //Haskell let in rule (let rec in exp + let in rule combined
        )
152     //gamma |- let rec f1 = e1 and f2 = e2 and f3 = e3 .... in e
        =>
```

```

153      //beta, [f1 -> tau1, f2 -> tau2, f3 -> tau3,...] + gamma |-
        e1 : tau1 | sigma1, [f1 -> sigma1(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
154  rule <k> typeInferenceFun(.ElemList, Gamma:Map, 'letIn(D:K,,
        E:K), Guess:KItem)
155      => typeInferenceFunLetIn(.ElemList, Gamma, makeDeclMap
        (D, TypeIt, .Map), D, E, Guess, 0, TypeIt, .Map,
        Beta) ...</k>
156      <typeIterator> TypeIt:Int => TypeIt +Int size(
        makeDeclMap(D, TypeIt, .Map)) </typeIterator>
157      <tempBeta> Beta:Map </tempBeta>
158
159  rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap
        :Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int
        , OldSigma:Map, Beta:Map)
160      => 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>
161      //=> typeInferenceFunLetIn(val(typeInferenceFun(
        DeclMap, grabLetDeclExp(D, Iter +Int TypeIt), Guess
        )), .ElemList, Gamma, DeclMap, D, E, Guess, Iter,
        TypeIt, OldSigma) ...</k>
162      requires Iter <Int (size(DeclMap))
163
164  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)
165      => typeInferenceFunLetIn(.ElemList, gammaSub(Sigma,
        Gamma,.Map), gammaSub(Sigma, DeclMap,.Map), D, E,
        typeSub(Sigma, Guess), Iter +Int 1, TypeIt, compose
        (Sigma,OldSigma), Beta) ...</k>
166      requires Iter <Int (size(DeclMap))
167

```

```
168     rule <k> typeInferenceFunLetIn(.ElemList, Gamma:Map, DeclMap
      :Map, D:KItem, E:KItem, Guess:KItem, Iter:Int, TypeIt:Int
      , OldSigma:Map, Beta:Map)
169       => typeInferenceFunLetIn(val (typeInferenceFun(.
          ElemList, Gamma applyGEN(Gamma, DeclMap, .Map, Beta
          ), E, Guess)), .ElemList, Gamma, DeclMap, D, E,
          Guess, Iter, TypeIt, OldSigma, Beta) ...</k>
170       requires Iter >=Int (size(DeclMap))
171
172     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)
173       => mapBagResult(compose(Sigma, OldSigma))...</k>
174       requires Iter >=Int (size(DeclMap))
175
176     rule mapLookup((Name |-> Type:KItem) DeclMap:Map, Name:KItem
      ) => Type
177     rule mapLookup(DeclMap:Map, Name:KItem) => Name
178       requires notBool(Name in keys(DeclMap))
179
180     //rule makeDeclMap('decls(A:K), TypeIt:Int, NewMap:Map) =>
      makeDeclMap(A, TypeIt +Int 1, NewMap)
181     rule makeDeclMap('decls(Dec:K), TypeIt:Int, NewMap:Map) =>
      makeDeclMap(Dec, TypeIt, NewMap)
182     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)])
183     rule makeDeclMap(.DeclList, TypeIt:Int, NewMap:Map) =>
      NewMap
184
185     rule grabLetDeclName('decls(Dec:K), Iter:Int) =>
      grabLetDeclName(Dec, Iter)
186     rule grabLetDeclName('declsList(Dec:K,, Rest:K), Iter:Int)
      => grabLetDeclName(Rest, Iter -Int 1)
187       requires Iter >Int 0
188     rule grabLetDeclName('declsList('declPatRhs('apatVar(Var:K)
      ,, Righthand:K),, Rest:K), Iter:Int) => Var
189       requires Iter <=Int 0
190
191
```



```
192     rule grabLetDeclExp('decls(Dec:K), Iter:Int) =>
      grabLetDeclExp(Dec, Iter)
193     rule grabLetDeclExp('declsList(Dec:K,, Rest:K), Iter:Int) =>
      grabLetDeclExp(Rest, Iter -Int 1)
194     requires Iter >Int 0
195     rule grabLetDeclExp('declsList('declPatRhs('apatVar(Var:K),,
      Righthand:K),, Rest:K), Iter:Int) => grabLetDeclExp(
      Righthand, Iter)
196     requires Iter <=Int 0
197     rule grabLetDeclExp('eqExpOptDecls(Righthand:K,, Opt:K),
      Iter:Int) => 'eqExpOptDecls(Righthand,, Opt)
198
199     rule genGamma('apatVar(Vari:K), Gamma:Map, Guess:K) => Gamma
      [Vari <- Guess]
200     rule genGamma('apatCon(Vari:K,, Pattwo:K), Gamma:Map, Guess:
      K) => Gamma[Vari <- Guess]
201
202     rule genLambda('apatVar(Vari:K), Ex:K) => Ex
203     rule genLambda('apatCon(Vari:K,, Pattwo:K), Ex:K) => '
      lambdaFun(Pattwo,, Ex)
204
205
206     rule gammaSub(Sigma:Map, (Key:KItem |-> Type:KItem) Gamma:
      Map, Newgamma:Map)
207     => gammaSub(Sigma, Gamma, Newgamma[Key <- typeSub(Sigma,
      Type) ] )
208     // => gammaSub(Sigma, Gamma, Newgamma[Key <- typeChildSub(
      Sigma, Type) ] )
209
210     rule gammaSub(Sigma:Map, .Map, Newgamma:Map)
211     => Newgamma
212
213     //rule typeChildSub((guessType(TypeIt) |-> Type:KItem) Sigma
      :Map, guessType(TypeIt:Int)) => Type
214
215     //rule typeChildSub(Sigma:Map, guessType(TypeIt:Int)) =>
      guessType(TypeIt)
216     //      requires notBool (guessType(TypeIt) in keys(Sigma))
217
218     rule freshInstance(guessType(TypeIt:Int), Iter:Int) =>
      guessType(TypeIt)
219     rule freshInstance(forAll(.Set, B:K), Iter:Int) => B
```

```
220     rule freshInstance(forAll(SetItem(C:KItem) A:Set, B:K), Iter
      :Int) => freshInstance(forAll(A, freshInstanceInner(C, B,
        Iter)), Iter +Int 1)
221
222     syntax KItem ::= freshInstanceInner(K,K,Int) [function]
223
224     rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K), Iter:
      Int) => funtype(freshInstanceInner(Repl,A,Iter),
        freshInstanceInner(Repl,B,Iter))
225     rule freshInstanceInner(Repl:KItem, Repl, Iter:Int) =>
      guessType(Iter)
226     rule freshInstanceInner(Repl:KItem, Target:KItem, Iter:Int)
      => Target [owise]
227
228     rule paramSize(forAll(A:Set, B:K)) => size(A)
229     rule paramSize(A:K) => 0 [owise]
230
231
232     rule applyGEN(Gamma:Map, (Key:KItem |-> Type:KItem) DeclMap
      :Map, NewMap:Map, Beta:Map)
233     => applyGEN(Gamma, DeclMap, NewMap[Key <- gen(Gamma, Type
      , Beta)], Beta)
234
235     rule applyGEN(Gamma:Map, .Map, NewMap:Map, Beta:Map)
236     => NewMap
237
238     //GEN
239     //GEN(Gamma, Tau) => Forall alpha
240
241     syntax KItem ::= gen(Map, K, Map) [function]
242     syntax Set ::= freeVarsTy(K, Map) [function]
243     syntax Set ::= freeVarsEnv(Map, Map) [function]
244     //syntax KItem ::= setBag(Set)
245     // syntax Set ::= listToSet(List, Set) [function]
246
247
248     rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
      => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
        freeVarsEnv(Gamma, Beta), Tau)
249     rule gen(Gamma:Map, Tau:KItem, Beta:Map) => forAll(
      freeVarsTy(Tau, Beta) -Set freeVarsEnv(Gamma, Beta), Tau)
      [owise]
250
```

```

251 //rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
      => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
      freeVarsEnv(Gamma, Beta), Tau)
252
253 rule freeVarsTy(guessType(TypeIt:Int), Beta:Map) => SetItem(
      guessType(TypeIt:Int))
254 rule freeVarsTy(funtype(Tauone:KItem, Tautwo:KItem), Beta:
      Map) => freeVarsTy(Tauone, Beta) freeVarsTy(Tautwo, Beta)
255 rule freeVarsTy(Tau:KItem, Beta:Map) => .Set
256     requires (forAll(.Set, Tau)) in values(Beta)
257 rule freeVarsTy(forAll(Para:Set, Tau:KItem), Beta:Map) =>
      freeVarsTy(Tau, Beta) -Set Para
258 rule freeVarsEnv(Gamma:Map, Beta:Map) => listToSet(values(
      Beta), .Set)
259
260
261 //      rule listToSet(ListItem(A:KItem) L:List, S:Set) =>
      listToSet(L, SetItem(A) S)
262 //      rule listToSet(.List, S:Set) => S
263
264 //////////////////////////////////////
265
266 //Unification
267
268 syntax Map ::= uniFun(List) [function]
269 //syntax List ::= uniSub(K,K,K) [function]
270 syntax Bool ::= isVarType(K) [function]
271 syntax Bool ::= notChildVar(K,K) [function]
272 syntax KItem ::= uniPair(K,K)
273
274 syntax List ::= uniSub(Map,K) [function] //apply
      substitution to unification
275
276 syntax KItem ::= typeSub(Map,K) [function] //apply
      substitution to type
277 syntax Map ::= compose(Map,Map) [function]
278
279 // syntax KItem ::= Map
280
281 rule uniFun(.List) => .Map //substi(.K,.K) is id
      substitution
282

```

```
283     rule uniFun(ListItem(uniPair(S:K,S)) Rest:List) => uniFun(
      Rest) //delete rule
284
285     // rule uniFun(SetItem(I:K)) => .K //uniFun(Rest) //delete
      rule
286
287     rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List) => uniFun(
      ListItem(uniPair(T,S)) Rest) //orient rule
288         requires isVarType(T) andBool (notBool isVarType(S))
289
290     //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
291     //   requires Svars !=K .List
292
293     //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
294
295     // 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
296     //   requires notChildVar('aexpQVar(Var:K),T)
297
298     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
299
300     rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List)
301         => compose((S |-> typeSub(uniFun(uniSub((S |-> T),Rest)),T
      )),uniFun(uniSub((S |-> T),Rest))) //eliminate rule
302     // => compose(uniFun(uniSub((S |-> T),Rest)),(S |-> typeSub
      (uniFun(uniSub((S |-> T),Rest)),T))) //eliminate rule
303         requires isVarType(S) andBool notChildVar(S,T)
304
305     rule isVarType(S:K) => true
306         requires getKLabel(S) ==KLabel 'guessType
307     rule isVarType(S:K) => false [otherwise]
308
309     rule notChildVar(S:K,T:K) => true
```

```
310
311     rule uniSub(Sigma:Map, .List) => .List
312     rule uniSub(.Map, L:List) => L
313     rule uniSub(Sigma:Map, Rest:List ListItem(uniPair(A:K, B:K))
314         ) => uniSub(Sigma, Rest) ListItem(uniPair(typeSub(Sigma,
315             A), typeSub(Sigma, B)))
316
317     //rule typeSub(substi(.Map), Tau:KItem) => Tau
318     rule typeSub(Sigma:Map (Tau |-> Newtau:KItem), Tau:KItem) =>
319         typeSub(Sigma (Tau |-> Newtau), Newtau)
320     rule typeSub(Sigma:Map, funtype(Tauone:KItem, Tautwo:KItem))
321         => funtype(typeSub(Sigma, Tauone), typeSub(Sigma, Tautwo))
322     rule typeSub(Sigma:Map, Tau:KItem) => Tau [owise]
323
324     syntax Map ::= composeIn(Map, Map, Map, K, K) [function]
325
326     rule compose(Sigmaone:Map, Sigmatwo:Map) => composeIn(
327         Sigmaone, Sigmatwo, .Map, .K, .K)
328
329     rule composeIn(Sigmaone:Map, (Key:KItem |-> Type:KItem)
330         Sigmatwo:Map, NewMap:Map, .K, .K) => composeIn(Sigmaone,
331         Sigmatwo, NewMap, Key, Type)
332
333     rule composeIn((Keyone |-> Typetwo:KItem) Sigmaone:Map,
334         Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) =>
335         composeIn(Sigmaone, Sigmatwo, NewMap, Keyone, Typeone)
336
337     rule composeIn((Typeone |-> Typetwo:KItem) Sigmaone:Map,
338         Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) =>
339         composeIn((Typeone |-> Typetwo) Sigmaone, Sigmatwo,
340         NewMap[Keyone <- Typetwo], .K, .K)
341         requires notBool(Keyone in keys(Sigmaone))
342
343     rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
344         Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
345         Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]
346
347     rule composeIn(Sigmaone:Map, .Map, NewMap:Map, .K, .K) =>
348         Sigmaone NewMap
349
350     //rule composeIn(Sigmaone:Map, .Map, .Map, .K, .K) =>
351         Sigmaone
```

```
337 //rule composeIn((Key:KItem |-> Type:KItem) Sigmaone:Map, .
    Map, NewMap:Map) => composeIn(Sigmaone, .Map, NewMap[Key
    <- mapLookup(Sigmaone, Type)])
338
339 //rule compose(Sigmaone:Map, Sigmatwo:Map) => updateMap(
    Sigmaone, Sigmatwo)
340 //rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:KItem)
    Sigmatwo:Map) => compose(Sigmaone[Keytwo <- Valtwo][
    Valtwo <- mapLookup(Sigmaone, Keytwo)], Sigmatwo)
341 //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
    Sigmatwo:Map, .K) => compose(Sigmaone[Type <- mapLookup(
    Sigmaone, Key)][Key <- Type], Sigmatwo, mapLookup(
    Sigmaone, Key))
342 //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
    Sigmatwo:Map) => composeIn(Sigmaone, Sigmatwo, mapLookup(
    Sigmaone, Key))
343 //      requires (notBool (Type in values(Sigmaone))) andBool
    (Type /=K mapLookup(Sigmaone, Key))
344 //rule compose(Sigmaone:Map, (Key:KItem |-> Type:KItem)
    Sigmatwo:Map) => compose(Sigmaone[Key <- Type][Type <-
    mapLookup(Sigmaone, Key)], Sigmatwo)
345 //      requires (notBool (Type in values(Sigmaone))) andBool
    (Type /=K mapLookup(Sigmaone, Key))
346 // rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:KItem)
    Sigmatwo:Map) => compose(Sigmaone[Valtwo <- mapLookup(
    Sigmaone, Keytwo)], Sigmatwo)
347 //      requires (Valtwo in values(Sigmaone)) andBool (Valtwo
    /=K mapLookup(Sigmaone, Keytwo))
348 //rule compose(Sigmaone:Map, (Keytwo:KItem |-> Valtwo:KItem)
    Sigmatwo:Map) => compose((Keytwo |-> Valtwo) Sigmaone,
    Sigmatwo)
349 // requires notBool (Keytwo in keys(Sigmaone))
350 //rule compose(Sigmaone:Map, .Map) => Sigmaone
351 // rule compose(substi(Sone:K, Tone:K), substi(Stwo:K, Ttwo:K))
    => substi(typeSub(substi(Stwo, Ttwo), Sone), Tone)
352
353
354 // rule notChildVar('aexpQVar(Var:K), T)
355
356
357 //T-Var
358 // rule typeInferenceFun('funApp(Eone:K,, Etwo:K), Alpha:Map,
    Beta:Map, Delta:Map, Gamma:Map, .K, .K) =>
```

```
359 //      typeInferenceFun('funApp(Eone,, Etwo), Alpha, Beta,
      Delta, Gamma,typeInferenceFun(Eone,Alpha, Beta, Delta, Gamma
      ,.K,.K),typeInferenceFun(Etwo,Alpha, Beta, Delta, Gamma,.K,.K
      ))
360 //      rule typeInferenceFun('funApp(Eone:K,, Etwo:K), Alpha:Map,
      Beta:Map, Delta:Map, Gamma:Map, funtype(Tauone:K, Tautwo:K),
      Tauone) => Tautwo
361
362      //T-Lam
363 //      rule typeInferenceFun('lambdaFun(Apatlist:K,, Ex:K), Alpha
      :Map, Beta:Map, Delta:Map, Gamma:Map,.K,.K) =>
364 //      typeInferenceFun('lambdaFun(Apatlist,, Ex), Alpha,
      Beta, Delta, Gamma,typeInferenceFun(Ex, Alpha, Beta, Delta,
      genGamma(Apatlist,Gamma),.K,.K),.K)
365
366 //      rule typeInferenceFun('lambdaFun(Apatlist:K,, Ex:K), Alpha
      :Map, Beta:Map, Delta:Map, Gamma:Map,Tautwo:K,.K) => Tautwo
367
368 endmodule
```

---

## REFERENCES