HASKELL SYNTAX AND STATIC SEMANTICS WRITTEN IN K-FRAMEWORK

Draft of November 28, 2018 at 21:56

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

To my parents, for their love and support.

Table of Contents

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.

The Haskell 2010 report gives a cursory description of the type system as a Standard Hindley Milner polymorphic type system, but gives no further indication of how this applies to the specifics of the Haskell syntax. In the specification in this paper, a more complete specification of parts of the type system is presented as a family of mutually inductive rules. Such a presentation is not only the basis of an executable semantics but can later provide the basis of formal and rigorous proofs.

By studying the standard Haskell compiler, GHC, one can see that there must be additional context sensitive checks that must be made prior to type inference.

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
           ][\+\-]?[0-9]+)?)
11                                    |([0-9]+[e E][\+\-]?[0-9]+)} [
                                         onlyLabel]
12     syntax CusChar ::= Token{[\'](~[\'\\\&])[\']} [onlyLabel]
13     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]
           | "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 where integers, and if 5 was an integer, or if the entire thing was one float. Normally in K, different tokens are separated with whitespaces. However, for some reason the parser had difficulty here. Initially, I added a workaround by requiring parentheses around each integer and floating point. 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 ImpDecl is an import declaration. An import is another module that this module depends on. 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 a name and a body of 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 and a body of 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.

The following is the definition of Decl and related sorts.

```
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)]
```

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

```
1   syntax GenDecl ::= VarsType
2                    | Vars "::" Context "=>" Type   [klabel('
                        genAssignContext)]
3                    | Fixity Ops
4                    | Fixity Integer Ops
5                    | "" [onlyLabel, klabel('emptyGenDecl)]
6
7   syntax FunLhs ::= Var APatList [klabel('varApatList)]
8                   | Pat VarOp Pat [klabel('patVarOpPat)]
9                   | "(" FunLhs ")" APatList [klabel('
                       funlhsApatList)]
10
11  syntax Rhs ::= "=" Exp OptDecls [klabel('eqExpOptDecls)]
12               | GdRhs OptDecls [klabel('gdRhsOptDecls)]
13
14  syntax GdRhs ::= Guards "=" Exp
15                 | Guards "=" Exp GdRhs
16  syntax Guards ::= "|" GuardList
17  syntax GuardList ::= Guard | Guard "," GuardList  [klabel('
       guardListCon)]
18  syntax Guard ::= Pat "<-" InfixExp
19                 | "let" Decls
20                 | InfixExp
```

```
21
22      //definition of exp
23      syntax Exp ::= InfixExp
24                   > 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 type inference function. This is because LExp defines the different expression types which the type inference function has specific rules for.

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

```
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 type inference function. The main parts of AExp that the inference function cares about is QVar and GCon.

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

```
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
```

2.4 Example Test Programs

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 the syntax data structure representing the code of the example program. They also contain the memory of the state machine. An actual state of the state machine in K is when the cells each have some term inside of them.

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 abstract syntax tree is initially placed into the $< k >$ cell. The command

```
1  $PGM:ModuleList
```

means that the parsed tree appears in this cell and the sort that contains all other sorts is ModuleList.

$.K$ means that the cell is initially empty.

tempModule is the name of the current module. tempCode is the current

code.

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

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

3.1 Alpha

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 $->$ 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.

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

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

3.2 Beta

tempT contains all user defined datatypes. tempT is organized in such a way that makes context sensitive checks easy to perform. tempBeta contains all user defined datatypes organized so that type inference is easy to perform. More is explained in chapter 5.

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

3.2.1 Example

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

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

Then the corresponding *tempBeta* should look like this. Note how the monomorphic datatype just has an empty $forAll$.

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

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

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

Then the corresponding tempBeta should look like this.

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

3.3 Delta

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

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

3.3.1 Example

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

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

Then the corresponding *tempDelta* should look like this.

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

3.4 Import Data Structure

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

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

3.5 Modules

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

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

```
18          </T>
19
20  endmodule
```

Chapter 4

Context Sensitive Checks

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

4.1 Data Types

Within the context of a computer program, a data type is a property of data that tells the compiler or interpreter more about how the data is supposed to be used within the context of the program. This is useful for having the compiler find bugs that occur from the programmer misusing data.

`https://en.wikipedia.org/wiki/Data_type`

In Haskell, a user can create a custom data type. This is referred to as a user defined type. Then when the user creates functions or any other expression, they can operate on their own data type.

4.1.1 Polymorphim

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

4.2 Datatype Constructors

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

Section 4 of the Haskell 2010 report specifies the haskell type system.

`https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-620004`

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

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

4.2.1 Type

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.

4.2.2 Data

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 type constructor *Date* followed by three integers.

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

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

4.2.3 Newtype

The third one is *newtype*,

```
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.

4.2.4 Context

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

4.3 Initial Data Structures

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

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

4.3.1 Example T

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

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

Then the corresponding *tempT* should look like this.

```
1     <tempT>
2         TList ( ListItem ( TObject ( Simp5 , CusBool , .TyVars ,
3             ListItem ( InnerTPiece ( True2 , .List , .List ,
                  True2
4               .OptBangATypes , CusBool ) )
```

```
5             ListItem ( InnerTPiece ( False2 , .List , .List ,
                  False2
6               .OptBangATypes , CusBool ) ) ) ) )
7     </tempT>
```

4.3.2 K Code

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

```
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(J:K) => getPreModule(J,preModule(AList(.
            List),TList(.List)))
30
31      rule getPreModule('bodytopdecls(I:K), J:K) => getPreModule(I
            ,J)
32      rule getPreModule('topdeclslist('type(A:K,, B:K),, Rest:K),J
            :K) => fetchTypes(A,B,Rest,J) //constructalpha
33
34
35      rule getPreModule('topdeclslist('data(A:K,, B:K,, C:K,, D:K)
            ,, Rest:K),J:K) => makeT(B,C,Rest,J)
36      rule getPreModule('topdeclslist('newtype(A:K,, B:K,, C:K,, D
            :K),, Rest:K),J:K) => makeT(B,C,Rest,J)
37
38
39      rule getPreModule('topdeclslist('topdecldecl(A:K),, Rest:K),
            J:K) => getPreModule(Rest,J)
40      rule getPreModule('topdeclslist('class(A:K,, B:K,, C:K,, D:K
            ),, Rest:K),J:K) => getPreModule(Rest,J)
41      rule getPreModule('topdeclslist('instance(A:K,, B:K,, C:K,,
            D:K),, Rest:K),J:K) => getPreModule(Rest,J)
42      rule getPreModule('topdeclslist('default(A:K,, B:K,, C:K,, D
            :K),, Rest:K),J:K) => getPreModule(Rest,J)
43      rule getPreModule('topdeclslist('foreign(A:K,, B:K,, C:K,, D
            :K),, Rest:K),J:K) => getPreModule(Rest,J)
44      rule getPreModule(.TopDecls,J:K) => J
45
46      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>
47            <tempModule> ModName:KItem </tempModule>
48
49      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>
50            <tempModule> ModName:KItem </tempModule>
51
```

```
52      rule makeInnerT(A:K,B:K,'nonemptyConstrs(C:K)) => makeInnerT
            (A,B,C)
53      rule makeInnerT(A:K,B:K,'singleConstr(C:K)) => ListItem(
            InnerTPiece(getCon(C),getTypeVars(C),getArgSorts(C),C,A))
54      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)
55
56      rule getTypeVars('constrCon(A:K,, B:K)) => getTypeVars(B)
57      rule getTypeVars('optBangATypes(A:K,, Rest:K)) =>
            getTypeVars(A) getTypeVars(Rest)
58      rule getTypeVars('optBangAType('emptyBang(.KList),, Rest:K))
             => getTypeVars(Rest)
59      rule getTypeVars('atypeGTyCon(A:K)) => .List
60      rule getTypeVars('atypeTyVar(A:K)) => ListItem(A)
61      rule getTypeVars(.OptBangATypes) => .List
62
63      rule getCon('constrCon(A:K,, B:K)) => A
64
65      rule getArgSorts('constrCon(A:K,, B:K)) => getArgSorts(B)
66      rule getArgSorts('optBangATypes(A:K,, Rest:K)) =>
            getArgSorts(A) getArgSorts(Rest)
67      rule getArgSorts('optBangAType('emptyBang(.KList),, Rest:K))
             => getArgSorts(Rest)
68      rule getArgSorts('atypeGTyCon(A:K)) => ListItem(A)
69      rule getArgSorts('atypeTyVar(A:K)) => .List
70      rule getArgSorts(.OptBangATypes) => .List
71
72  ////////////////////////////////////////////////////////////////////////////////

73
74      rule <k> preModule(A:K,T:K) => startTTransform ...</k>
75           <tempAlpha> OldAlpha:K => A </tempAlpha>
76           <tempT> OldT:K => T </tempT>
```

4.4 Context Sensitive Checks

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

K Code

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

```
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
                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
       on left. RIGHT SIDE ONLY
19 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
20
21    rule <k> startChecks
22             => checkNoSameKey
23                ~> (checkTypeConsDontCollide
24                ~> (makeAlphaMap
25                ~> (checkAlphaNoLoops
26                ~> (checkArgSortsAreTargets
27                ~> (checkParUsed))))) ...</k>
```

4.4.1 Name Collision

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

The following example should not be allowed.

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

```
2 ;type Date = Contwo Int
```

K Code

The following is the code for the check.

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

4.4.2 Type Constructor Collision

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

The following example should not be allowed.

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

The following example should also not be allowed.

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

K Code

The following is the code for the check.

```
    syntax KItem ::= keyCheck(K,K,K,K) [function] //(Alpha, T,
       List of names, Set of names)

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

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


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

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

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

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

4.4.3 Alpha Cycle Check

There should be no cycles in type renaming using *type*, and the type renaming chains using *type* should terminate with a type defined with *data* or *newtype*.

The following example should not be allowed.

```
1 type Birthday = Date
2 ;type Date = Birthday
```

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

```
1 type Birthday = Date
```

K Code

The following is the code for the check.

```
1     syntax KItem ::= aloopCheck(K,K,K,K,K,K,K) [function] //(
          Alpha,List of Alpha,Set of Alpha,CurrNode,lengthcheck,T,
          BigSet)
2
3     rule <k> checkAlphaNoLoops
4               => aloopCheck(A,.List,.Set,.K,.K,T,.Set) ...</k>
5          <tempAlphaMap> A:K </tempAlphaMap>
6          <tempT> T:K </tempT>
7
8     //aloopCheck set and list to check cycles
9     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)
10    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)
11
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(.Map, .List, .Set, .K, .K,T:K, S:Set) => .K
```

4.4.4 Argument Sort Check

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

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

```
1  Data Birthday = Birthday Date
```

K Code

The following is the code for the 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,
           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)
```

```
11          requires ((Arg in keys(AlphaMap)) orBool (Arg in Tset))
12
13      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
14          requires (notBool ((Arg in keys(AlphaMap)) orBool (Arg
               in Tset)))
15
16      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)
17
18      rule argSortCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,.
           List)) TListRest:List),AlphaMap:Map,Tset:Set) =>
           argSortCheck(TList(TListRest),AlphaMap,Tset)
19
20      rule argSortCheck(TList(.List),AlphaMap:Map,Tset:Set) => .K
```

4.4.5 Polymorphic Parameter Check

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

The following example should not be allowed.

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

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

The following example should not be allowed.

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

However, the following example should be allowed.

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

K Code

The following is the code for the 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 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))
13
14     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) =>
15          parCheck(TList(ListItem(TObject(ModName,A,B,ListItem(
                InnerTPiece(C,ParRest,D,E,F)) InnerRest)) Rest),
                NewSet)
16              requires Par in NewSet
17
18     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
19              requires notBool (Par in NewSet)
20
21     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) =>
22          parCheck(TList(ListItem(TObject(ModName,A,B,InnerRest))
                Rest),NewSet)
23
24     rule parCheck(TList(ListItem(TObject(ModName:K,A:K,B:K,.List
           )) Rest:List),NewSet:Set) =>
```

```
25          parCheck(TList(Rest),NewSet)
26
27      rule parCheck(TList(.List),NewSet:Set) => .K
28
29      rule makeTyVarList('typeVars(A:K,,Rest:K),NewList:List,
           NewSet:Set) => makeTyVarList(Rest, ListItem(A) NewList,
           SetItem(A) NewSet)
30
31      rule makeTyVarList(.TyVars,NewList:List,NewSet:Set) =>
           lengthRet(size(NewList),size(NewSet),NewSet)
32
33      rule lengthRet(A:Int, B:Int, C:K) => C
34                      requires A ==Int B
35
36      rule lengthRet(A:Int, B:Int, C:K) => error
37                      requires A =/=Int B
```

Chapter 5

Inferencing

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

[haskell 2010 report]

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.

`https://en.wikipedia.org/wiki/Type_system`

The type inference function infers the type of the expression that is fed into it. A purpose of it is to ensure that all functions and function applications are allowed with regards to the types of the inputs and outputs.

5.1 Type theory

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

`https://plato.stanford.edu/entries/type-theory/`

5.2 Data Structures

A monomorphic data type looks like $(a \to b) \to c$

In my K semantics, the equivalent data type looks like

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

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

In my K semantics, the equivalent data type looks like

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

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 inference.

5.2.1 Transform T into Beta

```
    // STEP 3 Transform T into beta

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

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

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

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

5.2.2 Construct Beta

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

Example

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

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

Then the corresponding *tempBeta* should look like this. Note how the monomorphic datatype just has an empty $forAll$.

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

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

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

Then the corresponding tempBeta should look like this.

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

K Code

The following is the K Code.

```
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
14      syntax KItem ::= betaParser(K,K,K) [function] //(Tree Piece,
           NewSyntax,Parameters,Constr)
15      syntax Set ::= getTyVarsRHS(K,List) [function]
16
17      syntax KItem ::= forAll(Set,K)
18      syntax KItem ::= funtype(K,K)
19
20      syntax Set ::= listToSet(List, Set) [function]
21
22      rule listToSet(ListItem(A:KItem) L:List, S:Set) => listToSet
           (L, SetItem(A) S)
23      rule listToSet(.List, S:Set) => S
24
25
26  //if optbangAtypes, need to see if first variable is a typecon
27  //if its a typecon then need to go into Delta and see the amount
        of parameters it has
28  //then count the number of optbangAtypes after the typecon
29      rule betaParser('constrCon(A:K,, B:K), Par:K, Con:K) =>
           forAll(getTyVarsRHS(B,.List), betaParser(B, Par, Con))
30      rule betaParser('optBangATypes('optBangAType('emptyBang(.
           KList),, 'atypeTyVar(Tyv:K)),, Rest:K), Par:K, Con:K) =>
           funtype(Tyv, betaParser(Rest, Par, Con))
31      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))
32      rule betaParser('optBangATypes('optBangAType('emptyBang(.
           KList),, 'atypeGTyCon(Tyc:K)),, Rest:K), Par:K, Con:K) =>
            funtype(Tyc, betaParser(Rest, Par, Con))
33      rule betaParser(.OptBangATypes, Par:K, Con:K) => '
           simpleTypeCon(Con,, Par)
34
```

```
35      rule getTyVarsRHS(.OptBangATypes,Tylist:List) => listToSet(
           Tylist, .Set)
36
37      rule <k> newBeta(M:Map)
38               => .K ...</k>
39           <tempBeta> OldBeta:K => M </tempBeta>
```

5.2.3 Construct Delta

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

Example

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

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

Then the corresponding *tempDelta* should look like this.

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

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

5.3 Definition of Substitution

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

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

5.4 Inference Algorithm

Much of the inference algorithm is the same as the one introduced in cs 421.

[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,
           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 ::= freshInstance(K, Int) [function]
20     syntax Int ::= paramSize(K) [function]
21
22
23     syntax KItem ::= mapBag(Map)
24     syntax KResult ::= mapBagResult(Map)
```

```
25
26      syntax Map ::= gammaSub(Map,Map,Map) [function]//
           substitution, gamma
27
28      rule <k> performIndividualInferencing => inferenceShell(Code
           ) ...</k>
29           <tempModule> Mod:KItem </tempModule>
30
31           <moduleName> 'moduleName(Mod) </moduleName>
32           <moduleTempCode> Code:KItem </moduleTempCode>
33
34      rule inferenceShell('topdeclslist('type(A:K,, B:K),, Rest:K)
           ) =>
35           inferenceShell(Rest) //constructalpha
36      rule inferenceShell('topdeclslist('data(A:K,, B:K,, C:K,, D:
           K),, Rest:K)) =>
37           inferenceShell(Rest)
38      rule inferenceShell('topdeclslist('newtype(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
39           inferenceShell(Rest)
40      rule inferenceShell('topdeclslist('class(A:K,, B:K,, C:K,, D
           :K),, Rest:K)) =>
41           inferenceShell(Rest)
42      rule inferenceShell('topdeclslist('instance(A:K,, B:K,, C:K
           ,, D:K),, Rest:K)) =>
43           inferenceShell(Rest)
44      rule inferenceShell('topdeclslist('default(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
45           inferenceShell(Rest)
46      rule inferenceShell('topdeclslist('foreign(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
47           inferenceShell(Rest)
48
49      rule inferenceShell('topdeclslist('topdecldecl(A:K),, Rest:K
           )) =>
50           typeInferenceFun(.ElemList, .Map,A,guessType(0)) ~>
                inferenceShell(Rest)
51
52
53      rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
           declFunLhsRhs(Fn:K,, Lhsrhs:K), Guess:K) =>
54           typeInferenceFun(.ElemList, Gamma, Lhsrhs, Guess) ...</
                k>
```

```
55      rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
           eqExpOptDecls(Ex:K,, Optdecls:K), Guess:K) =>
56           typeInferenceFun(.ElemList, Gamma, Ex, Guess) ...</k>
```

5.4.1 Variable Rule

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

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

5.4.2 Constant Rule

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

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

5.4.3 Lambda Rule

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

```
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,funtype(Tauone,Tautwo)))))
                ,Sigma)) ...</k>
```

5.4.4 Application Rule

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

```
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>
```

5.4.5 IfThenElse Rule

For functions, function guards, cases, and if-then-else are all equivalent.

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

```
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>
```

5.5 Mutual Recursion

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.

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.

5.5.1 LetIn Rule

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

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

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

$$\frac{\sigma'_0 = \{\} \qquad \text{dom}(\Psi) = \{\,x_1, \cdots, x_n\,\} \qquad \sigma_{i+1}(\Psi + \Gamma) \vdash e_i : \Psi(x_i) \,|\, \sigma_j \qquad \sigma'_i = \sigma_i \circ \sigma'_{j+1}}{\Gamma \vdash \{\,x_1 = e_1; \cdots x_i = e_i; \cdots x_n = e_n\,\}\,\text{in}\,\text{exp} : \bigcup_{i=1}^{n} \{\,x_i \mapsto \text{GEN}\,(\sigma'_n(\Gamma), \sigma'_n(\psi_i))\,\} \,|\, \sigma'_n}\ \text{LetIn}$$

K Code

```
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 -> simga1(tau1), f2 -> sigma1(
          tau2), f3 -> sigma1(tau3),....] + sigma1(gamma) |- e2 :
          sigma1(tau2) | sigma2  [f1 -> sigma2 o sigma1(tau1), f2
          -> sigma2 o sigma1(tau2), f3 -> sigma2 o sigma1(tau3)
          ,....] + sigma2 o sigma1(gamma) |- e3 : sigma2 o sigma1(
          tau3) .....  [f1 -> gen(sigma_n o sigma2 o sigma1(tau1),
          sigma_n o sigma2 o sigma1(Gamma)), f2 -> gen(tau2), f3 ->
          gen(tau3),....] + gamma |- e : something
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(.DeclsList, 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
```

5.6 Fresh Instance

```
1       rule freshInstance(guessType(TypeIt:Int), Iter:Int) =>
            guessType(TypeIt)
2       rule freshInstance(forAll(.Set, B:K), Iter:Int) => B
3       rule freshInstance(forAll(SetItem(C:KItem) A:Set, B:K), Iter
            :Int) => freshInstance(forAll(A, freshInstanceInner(C, B,
             Iter)), Iter +Int 1)
4
5       syntax KItem ::= freshInstanceInner(K,K,Int) [function]
6
7       rule freshInstanceInner(Repl:KItem, funtype(A:K, B:K), Iter:
            Int) => funtype(freshInstanceInner(Repl,A,Iter),
            freshInstanceInner(Repl,B,Iter))
8       rule freshInstanceInner(Repl:KItem, Repl, Iter:Int) =>
            guessType(Iter)
9       rule freshInstanceInner(Repl:KItem, Target:KItem, Iter:Int)
            => Target [owise]
10
11      rule paramSize(forAll(A:Set, B:K)) => size(A)
12      rule paramSize(A:K) => 0 [owise]
```

5.7 GEN

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

where

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

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

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

where c is a type such as Int

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

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

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

```
1       rule applyGEN(Gamma:Map, (Key:KItem |-> Type:KItem) DeclMap
            :Map, NewMap:Map, Beta:Map)
2         => applyGEN(Gamma, DeclMap, NewMap[Key <- gen(Gamma, Type
              , Beta)], Beta)
3
4       rule applyGEN(Gamma:Map, .Map, NewMap:Map, Beta:Map)
5         => NewMap
6
7      //GEN
8      //GEN(Gamma, Tau) => Forall alpha
9
10     syntax KItem ::= gen(Map, K, Map) [function]
11     syntax Set ::= freeVarsTy(K, Map) [function]
12     syntax Set ::= freeVarsEnv(Map, Map) [function]
13
14
15     rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
          => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
          freeVarsEnv(Gamma, Beta), Tau)
16     rule gen(Gamma:Map, Tau:KItem, Beta:Map) => forAll(
          freeVarsTy(Tau, Beta) -Set freeVarsEnv(Gamma, Beta), Tau)
           [owise]
17
18     //rule gen(Gamma:Map, forAll(Para:Set, Tau:KItem), Beta:Map)
           => forAll(freeVarsTy(forAll(Para:Set, Tau), Beta) -Set
          freeVarsEnv(Gamma, Beta), Tau)
19
20     rule freeVarsTy(guessType(TypeIt:Int), Beta:Map) => SetItem(
          guessType(TypeIt:Int))
```

```
21      rule freeVarsTy(funtype(Tauone:KItem, Tautwo:KItem), Beta:
           Map) => freeVarsTy(Tauone, Beta) freeVarsTy(Tautwo, Beta)
22      rule freeVarsTy(Tau:KItem, Beta:Map) => .Set
23           requires (forAll(.Set, Tau)) in values(Beta)
24      rule freeVarsTy(forAll(Para:Set, Tau:KItem), Beta:Map) =>
           freeVarsTy(Tau, Beta) -Set Para
25      rule freeVarsEnv(Gamma:Map, Beta:Map) => listToSet(values(
           Beta), .Set)
```

5.8 Unification Algorithm

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

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

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

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

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

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

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

Let $\phi = x \mapsto t$

Let ψ = Unif($\phi(S')$)

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

Note:

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

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(
           Rest)  //delete rule
16
17      rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List) => uniFun(
           ListItem(uniPair(T,S)) Rest) //orient rule
18        requires isVarType(T) andBool (notBool isVarType(S))
19
20      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
21
22      rule uniFun(ListItem(uniPair(S:K,T:K)) Rest:List)
23        => compose((S |-> typeSub(uniFun(uniSub((S |-> T),Rest)),T
             )),uniFun(uniSub((S |-> T),Rest))) //eliminate rule
24      //  => compose(uniFun(uniSub((S |-> T),Rest)),(S |-> typeSub
           (uniFun(uniSub((S |-> T),Rest)),T))) //eliminate rule
25        requires isVarType(S) andBool notChildVar(S,T)
26
27      rule isVarType(S:K) => true
28        requires getKLabel(S) ==KLabel 'guessType
29      rule isVarType(S:K) => false [owise]
30
31      rule notChildVar(S:K,T:K) => true
32
33      rule uniSub(Sigma:Map,.List) => .List
34      rule uniSub(.Map,L:List) => L
35      rule uniSub(Sigma:Map, Rest:List ListItem(uniPair(A:K, B:K))
           ) => uniSub(Sigma, Rest) ListItem(uniPair(typeSub(Sigma,
           A), typeSub(Sigma, B)))
36
37      rule typeSub(Sigma:Map (Tau |-> Newtau:KItem),Tau:KItem) =>
           typeSub(Sigma (Tau |-> Newtau),Newtau)
```

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

5.9 Composition of Substitutions

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

```
1       syntax Map ::= composeIn(Map, Map, Map, K, K) [function]
2
3       rule compose(Sigmaone:Map, Sigmatwo:Map) => composeIn(
           Sigmaone, Sigmatwo, .Map, .K, .K)
4
5       rule composeIn(Sigmaone:Map, (Key:KItem |-> Type:KItem)
           Sigmatwo:Map, NewMap:Map, .K, .K) => composeIn(Sigmaone,
           Sigmatwo, NewMap, Key, Type)
6
7       rule composeIn((Keyone |-> Typetwo:KItem) Sigmaone:Map,
           Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) =>
            composeIn(Sigmaone, Sigmatwo, NewMap, Keyone, Typeone)
8
9       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)
10           requires notBool(Keyone in keys(Sigmaone))
11
12      rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
           Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
           Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]
13
14      rule composeIn(Sigmaone:Map, .Map, NewMap:Map, .K, .K) =>
           Sigmaone NewMap
15  endmodule
```

5.10 Examples

Chapter 6

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
        lengthOfList(L)
18  //    rule lengthOfList(valValue(K:K),L:ElemList) => 1 +Int
        lengthOfList(L)
19
20      syntax Element ::= val(K) [strict]
21      syntax ElementResult ::= valValue(K)
22      syntax Element ::= ElementResult
23      syntax KResult ::= ElementResult
24      rule val(K:KResult) => valValue(K) [structural]
25
26      //form ElemList
27  //    syntax ElemList ::= formElemList(K) [function]
28
29      //CONVERT ~> TO List
30      //list convert
31  //    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, '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
```

6.1 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,
            B:KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
55            requires Node =/=K A
56
57
58      rule cycleCheck(none, M:Map, .List, L:List) => .K
59      rule cycleCheck(.K, M:Map, .List, I:List) => cycleCheck(
            getUnvisitedNode(.K, .K, M), M, .List, I)
60      rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:List)
            => cycleCheck(Node, M, S, I)
61      rule cycleCheck(Node:K, M:Map, S:List, I:List)
62                    => cycleCheck(.K, M[Node <- visited],
                          getNodeNeighbors(Node,I) S, I)
63            requires Node =/=K .K andBool Node =/=K none
64      rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
            cycleCheck(Node, M, S, I)
65            requires S =/=K .List
66
67  /*
68      rule cycleCheck(A:K,.K,.K,I:K) => cycleCheck(A,
            createVisitMap(I,.Map),.List,I)
69
70
71
72      rule cycleCheck(Node:K, M:Map, S:List, I:K) => cycleCheck(.K
            , M[Node <- visited], getNodeNeighbors(Node,I) S, I)
73
74      rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
            cycleCheck(Node, M, S, I)
75      //rule cycleCheck(.K, M:Map, .K, ListItem(impObject(A:K,B:K)
            ) Rest:List) => cycleCheck(ListItem(impObject(A:K,B:K))
            Rest:List)
76  */
```

6.2 Leaf DFS

```
1  //COPY IMPORT GRAPH, NEED SECOND GRAPH FOR RECURSING, ADDITIONAL
       GRAPH FOR SELECTING IMPORTS FOR ALPHA* AND BETA*
2  //DFS for leaf
3  //acquire alpha and beta for leaf
4  //merge alpha and beta with imports to produce alpha* and beta*
5  //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                                   ~> (
                                        performIndividualInferencing
                                        ))...</k>
23
24 //rule <k> dealWithImports(Mod:K, 'bodytopdecls(A:K)) =>
       callInit(A) ...</k>
25
26 //    rule <k> leaf
27 //             => cycleCheck(.K,createVisitMap(I, .Map),.List,I)
        ...</k>
28 //         <importTree> I:List </importTree>
29
30 ////////////////////////////////////////////////////////////////////////////////

31
32     syntax KItem ::= returnLeafDFS(K,List,Map) [function] //
          current node, map of all nodes to visited or not, stack,
          graph
33     syntax KItem ::= innerLeafDFS(K,List) [function]
34     syntax KItem ::= loadModule(K)
35
36     rule <k> leafDFS
37              => returnLeafDFS(.K,I,M) ...</k>
38          <recurImportTree> I:List </recurImportTree>
39          <impTreeVMap> M:Map </impTreeVMap>
40
```

```
41      rule returnLeafDFS(.K,ListItem(impObject(Node:KItem,B:KItem)
           ) I:List,M:Map) => returnLeafDFS(B,I,M)
42      rule returnLeafDFS(Node:KItem,I:List,M:Map) => returnLeafDFS
           (innerLeafDFS(Node,I),I,M)
43        requires innerLeafDFS(Node,I) =/=K none
44      rule returnLeafDFS(Node:KItem,I:List,M:Map) => loadModule(
           Node)
45        requires innerLeafDFS(Node,I) ==K none
46
47      rule innerLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem
           )) I:List) => B
48      rule innerLeafDFS(Node:KItem,ListItem(impObject(A:KItem,B:
           KItem)) I:List) => innerLeafDFS(Node,I)
49        requires Node =/=K A
50      rule innerLeafDFS(Node:KItem,.List) => none
51  //    returnLeafDFS(Node:KItem,ListItem(impObject(Node,B:KItem))
       I:List,M:Map) => returnLeafDFS(B,I,M)
52
53
54  ////////////////////////////////////////////////////////////////////////////////

55
56      //call before Checker Code
57  //    rule <k> callInit(S:K) => initPreModule(S) ...</k>
58  //         <tempModule> A:K => S </tempModule>
59
60      rule <k> loadModule(S:KItem) => .K ...</k>
61           <tempModule> A:K => S </tempModule>
62
63      rule <k> getAlphaAndBeta => initPreModule(Code) ...</k>
64           <tempModule> Mod:KItem </tempModule>
65
66           <moduleName> 'moduleName(Mod) </moduleName>
67           <moduleTempCode> Code:KItem </moduleTempCode>
```

6.3 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 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
           ][\+\-]?[0-9]+)?)
11                                    |([0-9]+[e E][\+\-]?[0-9]+)} [
                                         onlyLabel]
12     syntax CusChar ::= Token{[\'](~[\'\\\&])[\']} [onlyLabel]
13     syntax CusString ::= Token{[\"](~[\"\\]*)[\"]} [onlyLabel]
14     syntax VarId ::= Token{[a-z\_][a-z A-Z\_0-9\']*} [onlyLabel]
           | "size" [onlyLabel]
15     syntax ConId ::= Token{[A-Z][a-zA-Z \_0-9\']*} [onlyLabel]
16     syntax VarSym ::= Token{
17     ([\! \# \$ \% \& \* \+ \/ \> \? \^][\! \# \$ \% \& \* \+ \.
          \/ \< \= \> \? \@ \\ \^ \| \- \~ \:]*)
18    |[\-] | [\.]
19    |([\.][\! \# \$ \% \& \* \+ \/ \< \= \> \? \@ \\ \^ \| \- \~
          \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \-
          \~ \:]*)
20    | ([\-][\! \# \$ \% \& \* \+ \. \/ \< \= \? \@ \\ \^ \| \~
          \:][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \~
          \:]*)
21    | ([\@][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \-
          \~ \:]+)
22    | ([\~][\! \# \$ \% \& \* \+ \. \/ \< \= \> \? \@ \\ \^ \| \-
          \~ \:]+)
```

```
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
                                  preprocessing to check
88                             //if the CQList is a cname list or a
                                  qvar list.
89                           | "" [onlyLabel, klabel('
                                emptyOptCNameList)]
90      syntax CQList ::= List{CQName, ","}
91
92      syntax ImpDecl ::= "import" OptQualified ModId OptAsModId
           OptImpSpec [klabel('impDecl)]
93                       | "" [onlyLabel, klabel('emptyImpDecl)]
94      syntax OptQualified ::= "qualified"
95                             | ""  [onlyLabel, klabel('
                                  emptyQualified)]
96      syntax OptAsModId ::= "as" ModId
97                            | ""    [onlyLabel, klabel('
                                emptyOptAsModId)]
98
99      syntax OptImpSpec ::= ImpSpec
100                           | ""    [onlyLabel, klabel('
                                emptyOptImpSpec)]
101
102     syntax ImpSpecKey ::= "(" ImportList OptComma ")"
103     syntax ImpSpec ::= ImpSpecKey
104                      | "hiding" ImpSpecKey
105
106     syntax ImportList ::= List{Import, ","}
107
108     syntax Import ::= Var
109                     | TyCon CQList
110     syntax TopDecls ::= List{TopDecl, ";"} [klabel('topdeclslist
           )]
```

```
111
112     syntax TopDecl ::= Decl [klabel('topdecldecl)]
113                      > "type" SimpleType "=" Type [klabel('type)
                             ]
114                      | "data" OptContext SimpleType OptConstrs
                            OptDeriving  [klabel('data)]
115                      | "newtype" OptContext SimpleType "="
                            NewConstr OptDeriving [klabel('newtype)]
116                      | "class" OptContext ConId TyVar OptCDecls
                            [klabel('class)]
117                      | "instance" OptContext QTyCon Inst
                            OptIDecls [klabel('instance)]
118                      | "default" Types [klabel('default)]
119                      | "foreign" FDecl [klabel('foreign)]
120
121     syntax FDecl ::= "import" CallConv CusString Var "::" FType
122                    | "import" CallConv Safety CusString Var "::"
                          FType
123                    | "export" CallConv Safety CusString Var "::"
                          FType
124     //Liyi: fdecl needs to use special function in preprocessing
125     // to get the actually elements from the impent and expent
           from the CusString
126     //did string analysis
127
128     syntax Safety ::= "unsafe" | "safe"
129
130     syntax CallConv ::= "ccall" | "stdcall" | "cplusplus" | "jvm
           " | "dotnet"
131     syntax FType ::= FrType
132                    | FaType "->" FType // unsure about this one
                          syntax is ambiguous UNFINISHED
133
134     syntax FrType ::= FaType
135                     | "()"
136
137     syntax FaType ::= QTyCon ATypeList
138
139     //define declaration.
140     syntax OptDecls ::= "where" Decls | "" [onlyLabel, klabel('
           emptyOptDecls)]
141     syntax Decls ::= "{" DeclsList "}" [klabel('decls)]
142     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
           ('emptyOptCDecls)]
149     syntax CDecls ::= "{" CDeclsList "}"
150     syntax CDeclsList ::= List{CDecl, ";"}
151
152     syntax CDecl ::= GenDecl
153                    | FunLhs Rhs
154                    | Var Rhs
155
156     syntax OptIDecls ::= "where" IDecls | "" [onlyLabel, klabel
           ('emptyOptIDecls)]
157     syntax IDecls ::= "{" IDeclsList "}"
158     syntax IDeclsList ::= List{IDecl, ";"} [klabel('ideclslist)]
159
160     syntax IDecl ::= FunLhs Rhs [klabel('cdeclFunLhsRhs)]
161                    | Var Rhs [klabel('cdeclVarRhs)]
162                    | "" [onlyLabel, klabel('emptyIDecl)]
163
164     syntax GenDecl ::= VarsType
165                      | Vars "::" Context "=>" Type   [klabel('
                            genAssignContext)]
166                      | Fixity Ops
167                      | Fixity Integer Ops
168                      | "" [onlyLabel, klabel('emptyGenDecl)]
169
170     //three optional data type for the TopDecl data operator.
171     //deriving data type
172     syntax OptDeriving ::= Deriving | "" [onlyLabel, klabel('
           emptyDeriving)]
173     syntax Deriving ::= "deriving" DClass
174                       | "deriving" "(" DClassList ")"
175     syntax DClassList ::= List{DClass, ","}
176     syntax DClass ::= QTyCon
177
178     syntax FunLhs ::= Var APatList [klabel('varApatList)]
179                     | Pat VarOp Pat [klabel('patVarOpPat)]
180                     | "(" FunLhs ")" APatList [klabel('
                           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     syntax Guards ::= "|" GuardList
188     syntax GuardList ::= Guard | Guard "," GuardList  [klabel('
           guardListCon)]
189     syntax Guard ::= Pat "<-" InfixExp
190                    | "let" Decls
191                    | InfixExp
192
193     //definition of exp
194     syntax Exp ::= InfixExp
195                  > InfixExp "::" Type  [klabel('expAssign)]
196                  | InfixExp "::" Context "=>" Type  [klabel('
                        expAssignContext)]
197
198     syntax InfixExp ::= LExp
199                       > "-" InfixExp   [klabel('minusInfix)]
200                       > LExp QOp InfixExp
201
202     syntax LExp ::= AExp
203                   > "\\" APatList "->" Exp [klabel('lambdaFun)]
204                   | "let" Decls "in" Exp [klabel('letIn)]
205                   | "if" Exp OptSemicolon "then" Exp
                         OptSemicolon "else" Exp [klabel('ifThenElse
                         )]
206                   | "case" Exp "of" "{" Alts "}" [klabel('caseOf
                         )]
207                   | "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     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
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 preprossing to
                                     check if the Atype list is
                                     empty
62                                //it must have at least one item
63
64      //define type and simple type
65      syntax SimpleType  ::= TyCon TyVars  [klabel('simpleTypeCon)
           ]
66      syntax Type ::= BType
67                    | BType "->" Type  [klabel('typeArrow)]
68      syntax BType ::= AType
69                     | BType AType [klabel('baTypeCon)]
70
71      syntax ATypeList ::= List{AType, ""} [klabel('atypeList)]
72
```

```
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                       | Pat "," PatTuple    [klabel('patTupleCon
                             )]
120     syntax PatList ::= Pat
121                       | Pat "," PatList     [klabel('patListCon)
                             ]
122
123     syntax FPats ::= List{FPat, ","}
124     syntax FPat ::= QVar "=" Pat
125
126     //definition of quals
127     syntax Quals ::= Qual | Qual "," Quals  [klabel('qualCon)]
128
129     syntax Qual ::= Pat "<-" Exp
130                   | "let" Decls
131                   | Exp
132
133     //definition of alts
134     syntax Alts ::= Alt | Alt ";" Alts
135
136     syntax Alt ::= Pat "->" Exp  [klabel('altArrow)]
137                  | Pat "->" Exp "where" Decls
138                  | "" [onlyLabel, klabel('emptyAlt)]
139
140     //definition of stmts
141     syntax Stmts ::= StmtList Exp OptSemicolon
142     syntax StmtList ::= List{Stmt, ""}
143     syntax Stmt ::= Exp ";"
144                   | 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
       lengthOfList(L)
18 //    rule lengthOfList(valValue(K:K),L:ElemList) => 1 +Int
       lengthOfList(L)
19
20     syntax Element ::= val(K) [strict]
21     syntax ElementResult ::= valValue(K)
22     syntax Element ::= ElementResult
23     syntax KResult ::= ElementResult
24     rule val(K:KResult) => valValue(K) [structural]
25
26     //form ElemList
27 //    syntax ElemList ::= formElemList(K) [function]
28
29     //CONVERT ~> TO List
30     //list convert
31 //    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, '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
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:
           KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
           ListItem(B)
142     rule getNodeNeighbors(Node:KItem,ListItem(impObject(A:KItem,
           B:KItem)) Rest:List) => getNodeNeighbors(Node, Rest)
143       requires Node =/=K A
144
145
146     rule cycleCheck(none, M:Map, .List, L:List) => .K
147     rule cycleCheck(.K, M:Map, .List, I:List) => cycleCheck(
           getUnvisitedNode(.K, .K, M), M, .List, I)
148     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:List)
           => cycleCheck(Node, M, S, I)
149     rule cycleCheck(Node:K, M:Map, S:List, I:List)
150                 => cycleCheck(.K, M[Node <- visited],
                       getNodeNeighbors(Node,I) S, I)
151       requires Node =/=K .K andBool Node =/=K none
152     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
           cycleCheck(Node, M, S, I)
153       requires S =/=K .List
154
155 /*
156     rule cycleCheck(A:K,.K,.K,I:K) => cycleCheck(A,
           createVisitMap(I,.Map),.List,I)
157
158
159
160     rule cycleCheck(Node:K, M:Map, S:List, I:K) => cycleCheck(.K
           , M[Node <- visited], getNodeNeighbors(Node,I) S, I)
161
162     rule cycleCheck(.K, M:Map, ListItem(Node:K) S:List, I:K) =>
           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(
            TObject(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
                  of keys of alpha and targets of T, (keys of T)
342     syntax KItem ::= "checkParUsed"
343 //NEED TO CHECK all the polymorphic parameters from right appear
        on left. RIGHT SIDE ONLY
344 //NEED TO CHECK UNIQUENESS FOR POLY PARAM ON LEFT SIDE ONLY
345
346 //    rule <k> preModule(A:K,T:K) => startChecks ...</k>
347 //         <tempAlpha> OldAlpha:K => A </tempAlpha>
348 //         <tempT> OldT:K => T </tempT>
349
350
351 /*    rule <k> performNextChecks
352               => checkUseVars
353                  ~> (checkLabelUses
354                  ~> (checkBlockAddress(.K)
355                  ~> (checkNoNormalBlocksHavingLandingpad(.K, TNS
                        -Set TES)
356                  ~> (checkAllExpBlocksHavingLandingpad(.K, TES)
357                  ~> (checkAllExpInFromInvoke(.K, TES)
358                  ~> (checkLandingpad
359                  ~> checkLandingDomResumes)))))) ...</k> */
360
```

```
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,
           List of Tycons,Set of Tycons)
375     syntax KItem ::= lengthCheck(K,K) [function]
376     //syntax KItem ::= tyConCollCheck(K,K,K) [function]
377     //syntax K ::= innerCollCheck(K) [function]
378     //syntax K ::= tyConCollCheckPasser(K, K) [function]
379
380     //rule tChecker(preModule(Alpha:Map,T:K,Mod:K)) =>
           tyConCollCheck(innerCollCheck(T),preModule(Alpha,T,Mod))
381
382     //rule tyConCollCheck(.K,preModule(Alpha:Map,H:K,Mod:K)) =>
           tyConCollCheck(innerCollCheck(H),preModule(Alpha,H,Mod))
383
384     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) =>
385                  tyConCollCheck(TList(ListItem(TObject(
                        ModName,A,B,Inners)) Rest),ListItem(Ty) J
                        , SetItem(Ty) D)
386     rule tyConCollCheck(TList(ListItem(TObject(ModName:K, A:K,B:
           K,.List)) Rest:List),J:List,D:Set) =>
387                  tyConCollCheck(TList(Rest),J,D)
388     rule tyConCollCheck(TList(.List),J:List,D:Set) =>
389                  lengthCheck(size(J),size(D))
390
391     rule lengthCheck(A:Int, B:Int) => .K
392                  requires A ==Int B
393
394     rule lengthCheck(A:Int, B:Int) => error
395                  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,
           C:K)) Rest:List),M:Map) =>
551          makeDelta(TList(Rest),M[ModPlusType(ModName,A) <- size(
                retPolyList(Polys,.List))])
552     rule makeDelta(TList(.List),M:Map) => newDelta(M)
553
554     rule retPolyList('typeVars(A:K,,Rest:K),NewList:List) =>
           retPolyList(Rest, ListItem(A) NewList)
555     rule retPolyList(.TyVars,L:List) => L
556
557     rule <k> newDelta(M:Map)
558             => .K ...</k>
559          <tempDelta> OldDelta:K => M </tempDelta>
560
561     rule <k> constructBeta
562             => makeBeta(T,.Map) ...</k>
563          <tempT> T:K </tempT>
564
565     syntax KItem ::= makeBeta(K,Map) [function] //(T,Beta,Delta)
566
567     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) =>
568          makeBeta(TList(ListItem(TObject(ModName,A,B,InnerRest))
                Rest),Beta[ModPlusType(ModName,Con) <- betaParser(E
               ,B,A)])
569     rule makeBeta(TList(ListItem(TObject(ModName:K,A:K,B:K,.List
           )) Rest:List),Beta:Map) =>
570          makeBeta(TList(Rest),Beta)
571     rule makeBeta(TList(.List),Beta:Map) =>
572          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
           ) ...</k>
30            <tempModule> Mod:KItem </tempModule>
31
32            <moduleName> 'moduleName(Mod) </moduleName>
33            <moduleTempCode> Code:KItem </moduleTempCode>
34
35      rule inferenceShell('topdeclslist('type(A:K,, B:K),, Rest:K)
           ) =>
36            inferenceShell(Rest) //constructalpha
37      rule inferenceShell('topdeclslist('data(A:K,, B:K,, C:K,, D:
           K),, Rest:K)) =>
38            inferenceShell(Rest)
39      rule inferenceShell('topdeclslist('newtype(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
40            inferenceShell(Rest)
41      rule inferenceShell('topdeclslist('class(A:K,, B:K,, C:K,, D
           :K),, Rest:K)) =>
42            inferenceShell(Rest)
43      rule inferenceShell('topdeclslist('instance(A:K,, B:K,, C:K
           ,, D:K),, Rest:K)) =>
44            inferenceShell(Rest)
45      rule inferenceShell('topdeclslist('default(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
46            inferenceShell(Rest)
47      rule inferenceShell('topdeclslist('foreign(A:K,, B:K,, C:K,,
            D:K),, Rest:K)) =>
48            inferenceShell(Rest)
49
50      rule inferenceShell('topdeclslist('topdecldecl(A:K),, Rest:K
           )) =>
51            typeInferenceFun(.ElemList, .Map,A,guessType(0)) ~>
                 inferenceShell(Rest)
52
53
54      rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
           declFunLhsRhs(Fn:K,, Lhsrhs:K), Guess:K) =>
55            typeInferenceFun(.ElemList, Gamma, Lhsrhs, Guess) ...</
                 k>
56      rule <k> typeInferenceFun(.ElemList, Gamma:Map, '
           eqExpOptDecls(Ex:K,, Optdecls:K), Guess:K) =>
57            typeInferenceFun(.ElemList, Gamma, Ex, Guess) ...</k>
58
```

```
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 -> simga1(tau1), f2 -> sigma1(
           tau2), f3 -> sigma1(tau3),....] + sigma1(gamma) |- e2 :
           sigma1(tau2) | sigma2  [f1 -> sigma2 o sigma1(tau1), f2
           -> sigma2 o sigma1(tau2), f3 -> sigma2 o sigma1(tau3)
           ,....] + sigma2 o sigma1(gamma) |- e3 : sigma2 o sigma1(
           tau3) .....  [f1 -> gen(sigma_n o sigma2 o sigma1(tau1),
           sigma_n o sigma2 o sigma1(Gamma)), f2 -> gen(tau2), f3 ->
            gen(tau3),....] + gamma |- e : something
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(.DeclsList, 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 [owise]
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))
            ) => uniSub(Sigma, Rest) ListItem(uniPair(typeSub(Sigma,
            A), typeSub(Sigma, B)))
314
315     //rule typeSub(substi(.Map),Tau:KItem) => Tau
316     rule typeSub(Sigma:Map (Tau |-> Newtau:KItem),Tau:KItem) =>
            typeSub(Sigma (Tau |-> Newtau),Newtau)
317     rule typeSub(Sigma:Map,funtype(Tauone:KItem,Tautwo:KItem))
            => funtype(typeSub(Sigma,Tauone),typeSub(Sigma,Tautwo))
318     rule typeSub(Sigma:Map,Tau:KItem) => Tau [owise]
319
320     syntax Map ::= composeIn(Map, Map, Map, K, K) [function]
321
322     rule compose(Sigmaone:Map, Sigmatwo:Map) => composeIn(
            Sigmaone, Sigmatwo, .Map, .K, .K)
323
324     rule composeIn(Sigmaone:Map, (Key:KItem |-> Type:KItem)
            Sigmatwo:Map, NewMap:Map, .K, .K) => composeIn(Sigmaone,
            Sigmatwo, NewMap, Key, Type)
325
326     rule composeIn((Keyone |-> Typetwo:KItem) Sigmaone:Map,
            Sigmatwo:Map, NewMap:Map, Keyone:KItem, Typeone:KItem) =>
             composeIn(Sigmaone, Sigmatwo, NewMap, Keyone, Typeone)
327
328     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)
329           requires notBool(Keyone in keys(Sigmaone))
330
331     rule composeIn(Sigmaone:Map, Sigmatwo:Map, NewMap:Map,
            Keyone:KItem, Typeone:KItem) => composeIn(Sigmaone,
            Sigmatwo, NewMap[Keyone <- Typeone], .K, .K) [owise]
332
333     rule composeIn(Sigmaone:Map, .Map, NewMap:Map, .K, .K) =>
            Sigmaone NewMap
334
335     //rule composeIn(Sigmaone:Map, .Map, .Map, .K, .K) =>
            Sigmaone
336
```

```
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