HASKELL SYNTAX AND STATIC SEMANTICS WRITTEN IN K-FRAMEWORK

Draft of October 27, 2018 at 19:58

BY

BRADLEY MORRELL

THESIS

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

Urbana, Illinois

Adviser:

Your Adviser Here

ABSTRACT

Replace this text with your concise (300 words or less) summary of thesis contents. Note that you can ignore the TOC on page iv because it is generated automatically. Work on the body of the thesis, then hit the Update tab on the TOC and voil. When you double-clicked ECE 499 template.dotx, you opened a new, untitled document in Microsoft Word, which has the main components of your thesis laid out for you. Save the new document, replace the red text and bracketed section heads with your own, insert carefully prepared graphics, follow the Guidelines document, proofread and revise, and youll likely end up with a successful thesis. You may wish to change the line spacing to double rather than 1.5 lines; make sure the spacing is consistent throughout the thesis.

Subject Keywords: at least one subject keyword here; separate keywords with semicolons

To my parents, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CHAPTER 1

INTRODUCTION

In the first progress report, I will detail what I learned about K and the motivation for specifying languages in K, along with Haskell and the goal of my project. A current problem with engineered systems is that the design of the system is not proven to be working. The system could be non-functional at design time. The designer may not fully understand the system environment or may have not considered the behavior of the system in rare circumstances. Then once the design is made, there may be bugs introduced by implementing the design incorrectly. The current way that programs are created is by making a design or a formal specification of the program, implementing it, and testing the program against unit tests or verifying the behavior of the program after the fact. Formal methods are ways to mathematically prove correctness of a system. Without formal methods, the only way to reason about a system is by testing it against different edge cases. Within the context of a programming language, one way a programming language can be formally specified is by defining a syntax and semantics for that language. The operational semantics of a programming language can be thought of as a transition system upon an abstract syntax tree, which is the program itself written in the language, and a state, which is a function from the variables in the tree to the current values of those variables. This way, real and complex programs written in natural looking programming languages can be interpreted as strings written in formal languages. Once a programming language is defined in this way, certain properties and behavior of the language and programs written in the language can be proven. K is a framework for creating the formal specification of a programming language. It then can interpret programs written in the language by running only the rules of the formal operational semantics of the programming language. This allows programs to be ran and analyzed formally. This way the formal specification of the complex programming language can be tested and analyzed with the

use of a machine. A K-configuration defines the memory structure of the programming language, made up of cells. The program state can be thought of as the current values of the K-configuration at a certain point in time. Grammar can be written in K using the constructor syntax, and a semantic rule can be written in K using the constructor rule. Haskell is a purely functional programming language with strong static typing. Purely functional means that the language only allows the user to make functions whose output is only dependant on the function input. Strong static typing means that before a program is run, a type inference algorithm checks the program and ensures that all functions and function applications are allowed with regards to the types of the inputs and outputs. Static refers to the fact that type inference is performed before the code is ran, and will not run during the runtime of the code. Strong typing refers to the fact that the compiler will not allow the user to perform workarounds like typecasting. My goal for this project is to write the syntax of Haskell and the type system of Haskell in K.

CHAPTER 2

CONTEXT FREE SYNTAX

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

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

Means that modid . is optional, and the brackets [ ] are not part of the haskell code, but the period . is part of the haskell code. Any symbol that is not in bold needs to be written in the program in order to parse correctly. There are a lot of parts of the grammar that were tricky for me to implement in K. For instance, a sort definition with an optional part could be just written using a — in the K syntax. So

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

Is written in my K syntax as

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

However, an issue arises when you have something written like

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

As an option for the definition of the topdecl sort. What I did was, for each optional part, replace the optional part with a new sort. For instance, I replaced [context =¿] with a new sort called OptContext. Then I just specified

```
syntax OptContext ::= Context "=>" | "" [onlyLabel, klabel('emptyContex
```

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 function application and pattern. So for functions f, y, and z, if we were to write f applied to y applied to z applied to 2, we would normally write... f y z 2 With the parentheses notation it looked like f (y (z (2))) This caused the programs to quickly look unreadable. So I instead opted to only require that integers and floats have parentheses surrounding them. f y z (2) This fixed the issue. I ran into another issue where a program with a variable called size did not parse. I found out that this is because size is a k keyword. So I just specified that a variable could be a variable token, or size.

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

CHAPTER 3

CONTEXT SENSITIVE CHECKS

I also placed the user defined types into data structures in order to perform several checks to make sure that the user did not have errors when creating types. Then the data structures will be transformed into a form that will be used for type inferencing. Section 4 of the Haskell 2010 report specifies the haskell type system. In the topdecl sort, there are three typecons that are used to create user defined datatypes. Data, type, and newtype. The end goal is to put the user defined types into a data structure which I can use to perform type inferencing. These three typecons are used to create user defined types. The first one is type, type simpletype = type This is used in a haskell program to declare a new type as a single type. In effect, it renames the type where both names now can be used to refer to the type. type Username = String Is one such example usage of type, it creates a new type Username, which is defined as just a string. Now the programmer can refer to Username or String to make a string. The second one is data,

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

This allows a user to declare a new type that may include many fields, and polymorphic types. For instance: data Date = Date Int Int Int This is a new type that includes the typecon Date followed by three integers. data Poly a = Number a This is a new polymorphic type with polymorphic parameter a, that has the typecon Number. The third one is newtype,

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

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

I perform several checks here, 1. The programmer should not be able to make two user defined datatypes with the same name, even if one is created using data and another is created using type for instance. 2. The programmer should not be able to use the same typecons when making different options

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

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

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

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.

CHAPTER 4

MULTIPLE MODULE SUPPORT

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

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

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

```
File 1:
module File1 where
{data A = B
}
```

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

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

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

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

syntax ModuleList ::= Module [klabel('modListSingle)] — Module "¡NEXTMODULE¿" ModuleList [klabel('modList)]

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

CHAPTER 5

INFERENCING

5.1 Data Structures

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

$$fx = yx$$

$$yx = fx$$

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

`[context =>]`

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

5.1.1 Inferencing Rules

$$\dfrac{\dfrac{\dfrac{\text{Hyp}}{\{A \wedge B\} \vdash A \wedge B}\quad\dfrac{\text{Hyp}}{\{A \wedge B\} \cup \{B\} \vdash B}}{\{A \wedge B\} \vdash B}\,\text{And}_R\ \text{E}\quad\dfrac{\dfrac{\dfrac{\text{Hyp}}{\{A \wedge B\} \vdash A \wedge B}\quad\dfrac{\text{Hyp}}{\{A \wedge B\} \cup \{A\} \vdash A}}{\{A \wedge B\} \vdash A}\,\text{And I}}{\{A \wedge B\} \vdash B \wedge A}\,\text{Imp I}}{\{\} \vdash A \wedge B \Rightarrow B \wedge A}$$

Haskell is a strong and static type system.

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

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

5.2 Lambda Calculus

5.3 Type theory

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

5.4 Hindley-Milner

5.5 Inferencing Algorithm

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

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

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

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

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

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

5.5.1 Unification Algorithm

Let S = (s 1 , t 1), (s 2 , t 2), ..., (s n , t n) be a unification problem. Case S = : Unif(S) = Identity function (ie no substitution) Case S = (s, t) S): Four main steps

Delete: if s = t (they are the same term) then Unif(S) = Unif(S) Decompose: if s = f(q 1 , ... , q m) and t =f(r 1 , ... , r m) (same f, same m!), then Unif(S) = Unif((q 1 , r 1), ..., (q m , r m) S) Orient: if t = x is a variable, and s is not a variable, Unif(S) = Unif ((x,s) S)

Eliminate: if s = x is a variable, and x does not occur in t (the occurs check), then Let = x — t Let = Unif((S)) Unif(S) = x — (t) o Note: x — a o y — b = y — (x — a(b) o x — a if y not in a

[cs 421 class notes]

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

Type inferencing

CHAPTER 6

CONCLUSION

This is my conclusion.