Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

Generation of Type Checking Code

Emma Tomlinson

2019-April-28

Supervisor: Jeremy L. Jacob

For my Mum

Acknowledgements

I give thanks to my supervisor, Jeremy Jacob, for his continued help during the process of writing this project. I would also like to give thanks to my friends and family who have supported (and coped) throughout the project.

Contents

List of Listings

List of Figures

Executive Summary

Type systems are used to reduce the number of incorrect programs that a compiler will accept and generate code for. They are often the first line of defence against stopping bugs from being introduced into programs. For many years type systems have had a formal notation to specify them. There are a small number of tools that will generate code for type checkers, some of which have designed their notation such that it is akin to the formal notation. However, the feature sets that these tools support vary by a large amount, and many of these tools can be complicated. We are therefore going to devise a notation that can describe type systems in a machine readable format. The notation should also be intuitive, such that compiler writers not familiar with the formal notation of type systems will still be able to understand the type checkers written in our notation.

The notation should be able to support simple languages from different programming paradigms, such as imperative and function paradigms. It should be able to support arithmetic and Boolean expressions, such as addition, equality, etc. There should be the facility to support the use and declaration of variables and subprograms. It should be able to support more complicated types than just integer and Boolean, such as array types and record types. However, given the time scale for the project we are not expecting to be able to type check object orientated languages, such as Java, and other languages of similar complexity, so only the language features discussed are required and any extra languages features are to be implemented if time permits.

The notation we have devised is similar to the formal notation of type systems. However, its syntax is devised in such a way that uses programming constructs that developers should be familiar with, such as an if-then statement to assert properties within the type system and a return statement that can be used to return information. The notation differs from the notation used in other tools in a number of ways. Many other notations force the user to use a particular set of tools to perform lexical analysis and syntax analysis. The notation we have devised, and subsequent tool, do not have this issue because we use the abstract syntax definition of the language to both represent the type system in our notation and to perform the type checking in the generated code. This provides the user with the ability to use any parser and lexer they wish. The one stipulation, is the parser and lexer must be written in Haskell, or a tool that generates Haskell,

this is because the target language, the output language of the generated code, is Haskell. The tool generates Haskell code so that it makes use of some of the high level language features in Haskell, which makes traversing abstract syntax trees trivial. The algorithms that we generate are based off of the algorithms discussed in Ranta [1]. The notation allows the user to write their own Haskell code so that they interact with the context. Interacting with the context is important because the context stores all of the information about variables, subprograms and it may also contain user defined type information. The user defines their own context so that the kinds of languages is not limited by the context. However, to help the user to define their context, a Haskell type class is provided to the user with function signatures for common actions that would be performed on the context. The functions in these type classes are functions to make an empty context, add a new block to the context, lookup an item and retrieve its type from the context, and add a new item to the context with a given type.

The notation and tool is able to represent and generate code for two example languages that have been used as test cases to prove that the notation and tool can type check all of the features that have been specified. These two languages are very different, one is a simple imperative language, the other is a simple functional programming language. This demonstrates that the kinds of languages that the tool can represent and generate code for is diverse. However, the notation is far from perfect, it cannot type check every type of language, in particular it is unable to type check languages similar to ML and Haskell that have polymorphic type systems. The code is generated is also not as efficient as it could be, especially in an error case. There are also a number of extensions could make for the representation of the context and its given functions.

Statement of Ethics

There are very few ethical considerations to make. We have not used any people or animals as part of the testing process for the tool. There have been no experiments performed during the development of the tool or in any of its subsequent testing. Any software that was used during the development of the tool or used as part of the discussion during this report has been credited to the relevant people, we have not claimed any of the software as our own.

1 Introduction

Programming languages have had type systems and type checkers dating back to Fortran[2]. Since then, type systems have gone through many changes and an academic body of research has developed out of type systems to explore what information they can represent[3]. Over the past few decades type systems for modern programming languages have grown in sophistication and as a consequence type checkers have grown more complicated. As software becomes more complex, the likelihood for errors occurring becomes higher. If there is an error in a type checker then there is the possibility of code containing errors being passed through the compiler successfully, whereas if the compiler were to behave correctly, such an error could have been caught.

This leads to the possibility of generating the code for the type checkers based upon the specification of the type systems in order to remove some of the complexity of developing the type checker. Therefore, a notation will be presented that can be used to represent a specification of a type system From this notation code can be generated that implements the specification it is representing. The notation should be simple to use and easy to understand, The simplicity of such a notation should reduce the number of errors that occur in these complicated pieces of software. It should also be able to generate code for a large variety of programming languages with many different language features. The following list is the list of required features that our notation should have or be able to represent:

- Simple notation similar to the current method of representing type systems in academic literature but that should also be easy to understand by people not familiar with the academia.
- Be able to generate code for many different languages, for example for imperative and function programming languages.
- Be able to generate code for simple arithmetic expressions and simple statements, i.e. an addition expression and loop statements.
- Be able to generate code for variables and subprograms (functions and procedures).
- Be able to represent more complex data types, ranging from array types and record types, to functions types (i.e. lambda expressions).

These required features should allow for the notation to represent many kinds of type systems. They should also allow for the notation, and corresponding generated code, to be sophisticated enough to type check most simple languages whilst also allow for the possibility for extensions to add more supported kinds of languages in the future. This list of required features is how we will check for the success of the project and the notation. However, there is another set of features that are desirable but not required features. For example, all things permitting, it should be able to generate code for and represent user defined types types and type synonyms (i.e. typedef's in C[4]). Another possible feature is to generate code for Hindley-Milner type inference[5], this would allow for the ability to type check languages similar to ML[6]. This is not a required feature but it would greatly expand the scope of what the desired notation would be capable or representing.

The next chapter will discuss the background of type systems and type checkers, along with some previous type checker generators and discuss their advantages and disadvantages. In chapter 3, the notation for representing type systems will be given, along with the code that we generate from that notation. The different features in the notation will be expanded upon, examples of how the notation can, and is intended to be, used will then be given. In chapter 4, it will be discussed whether or not the notation was able to successfully represent and generate code for all of the required features. Then, there will be a discussion of any potential problems with the notation or with the code that is generated. Finally any further work that can be done to extend the notation or the code generated will be presented, the possible usefulness of such extensions will also be discussed.

2 State of the Art

The idea of generating code to be a part of the compiler pipeline is not new. The first two stages of the front-end of a compiler, the lexical analyser and the syntax analyser (more commonly known as the lexer and the parser), have general tools to generate their code using *Domain Specific Languages* (DSL)[7], [8]. Two of these languages are Flex[9], as the lexer generator, and Bison[9], as the parser generator. These tools generate C code to be used to build the rest of a compiler, possibly including a type checker and machine code generation. These programs have been around for many years, and many derivatives of them have been made to be used with different general purpose programming languages, such as Alex and Happy for Haskell or Jlex and Cup for Java[1]. Many languages have been made using these programs and their respective DSL's, for example the parser for Haskell is self-generated by a grammar defined in a piece of Happy code. Although, not all parts of the compiler pipeline have general tools that are widely used, some such tools do exists[10]–[12].

2.1 Type Systems

The concept and implementation of type systems has existed for almost as long as the concept of higher level language to perform computation[2]. Many languages have a type system in one form or another with varying degrees of complexity, ranging from languages with no type system (such as Scheme or JavaScript) to relatively simple type systems in languages such as C, and finally fairly complex and expressive type systems such as those in Haskell and Agda.

"A type system specifies the type rules of a programming language independently of particular type checking algorithms"[3], where type rules define under what circumstances a code fragment can be considered type correct. This is an important distinction as it states that a type system is not how to type check the language, but instead states that the type checker should confirm to the type rules specified by the type system. Therefore it is the type system that defines the behaviour of the type checker.

Rule Label: $\frac{J_1 \; J_2 \; .. \; J_n}{C} \; [Side \; Conditions]$

Figure 2.1: General form of a natural deduction rule

2.1.1 Usefulness of Type Systems

There are many arguments as to the usefulness of type systems and whether or not languages should have static type systems. All programs in a static type system can be checked before runtime, where as programs in a dynamic type systems can only be checked at runtime. An argument can be made that software could be made safer with the use of type systems. One of the most famous software bugs in history is that of the Ariane 5, resulting in the loss of the spacecraft after 39 seconds of flight. The bug that caused the loss of the Ariane 5 was in a function that converted a 64-bit floating point number to a 16-bit integer number. The bug was created when the function tried to convert the floating point number into a value that was larger than could be represented by a 16-bit integer[13]. A more sophisticated type system could have possibly identified the differences in what the two data types could represent and reported an error to the developers during compilation to tell them about the possibility of an unsafe conversion. This would have enabled the bug to have been identified and fixed before the launch of the spacecraft.

2.1.2 Type Rules

Much like how there is a formal representation for context free grammars as Backus-Naur form[1], [14], [15], there is also have a formal way of representing type systems and type rules. In the literature, the standard method of representing type systems and type rules is through the use of natural deduction rules[1], [3].

Natural deduction rules have a form of: given some premises we conclude the consequent[1], [16], for example in Figure 2.1 the premises are denoted as J_k and the consequent is denoted as C.

These natural deduction rules, can then be used, to specify the type rules in a given type system[1], [3]. Figure 2.2 is an example of how a simple type rule for an arithmetic expression using the natural deduction logic would be written. It is important to break this rule down, into its constituent parts to understand what it is trying to specify. Starting from the consequent $\Gamma \vdash e_1 + e_2 : Int$, Γ is the current context which we are applying the rule to. The $\vdash$ is piece of syntax to say given the left hand side the right hand side can be proven. Finally for the consequent $e_1 + e_2 : Int$ means that the

$$\text{Add Expression:} \quad \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 + e_2 : Int}$$

Figure 2.2: Simple arithmetic expression type rule

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : Bool} \, [\tau \in \{Bool, Int\}]$$

Figure 2.3: Type rule for equality for the example language for Eli

piece of concrete syntax $e_1 + e_2$ must have type Int. The top half of the rule are the things that must hold for the bottom half to be valid. So simply e_1 must be of type Int given the context Γ, and the same is for the second premise[1], [3].

2.2 Type Checker Generators

There have been multiple attempts to make the writing of type checkers easier on the compiler writers by creating new tools or notations. These can have very different ways of expressing type systems and what kinds of notations they use. Some use the notation of attribute grammars to represent type systems, along with the rest of the compiler, whereas other use more specialised DSL's to represent their type rules and type checkers.

2.2.1 Attribute Grammars

Attribute Grammars are a way of specifying the semantics of grammar rules for a Context Free Grammar. This makes them a likely candidate to look at when attempting to generate type checking code since type checking is a form of semantic analysis on some input code. In Knuth's paper[17], he gives an example of the attributes required to evaluate a simple language to emulate a Turing Machine. Even though he discusses the evaluation of a turing machine, they are not limited to expressing the semantics of evaluation, we can also use them to express the semantics of type systems such that on each node of the tree an attribute could be given to it that defines the type of the node.

2.2.2 Eli

Eli is a tool which, given a set of specifications, will generate code for a whole compiler based upon those specifications[12]. The way Eli generates all the code is by the use of an attribute grammar system and numerous internal tools that generate code for each different part of the compiler pipeline. The internal tools use DSL's which define a specification for part of the language. One of these DSL's is called *Operator Identification Language* (OIL), which is used to define the typing relations on operators. The definition of these type rules is fairly simple, however in their example there are a lot of repeated definitions for operators. For example there are multiple definitions to define the equality operator, one for each of the required types that the operator can act on. Where as in the natural deduction rules this would only have to be defined once where the type is some arbitrary type τ with some constraint on τ such that it can only be from a set of types that an equality check can be performed on, see the natural deduction rule given in Figure 2.3. The tools lacks built-in support for constructs in the language that can have arbitrary types. However, OIL is a small and simple language that is easy to understand, but, the other constituent parts of ELI required to build a tool are not as simple and the specifications written in them can get significantly more complicated for larger projects.

2.2.3 Ruler

Ruler is a DSL that bears close resemblance to natural deduction rules that formally define type systems[11]. The notation used to describe the type rules in ruler is intuitive to anyone who has experience with using natural deduction rules to represent type rules. However, other possible compiler writers may have never come across the notation therefore the syntax of Ruler could be un-intuitive to them. The rules can be compiled to either LaTeXor to an attribute grammar system in Haskell. However, this means that rules have to be written multiple times for each target that the user may want to output to. This creates unnecessary duplication of types rules since if a user wants both the executable code and the formal specification they will have to write the type system twice for both of these outputs. Since the only executable code that is generated compiles to an attribute grammar, it forces the user to use the attribute grammar for the whole frontend of their compiler, whereas using the abstract syntax allows the user to choose any parser generator (or write their own parser) as long they provide the abstract syntax to the tool. They also make a claim that they support Hindley-Milner type inference. While this to some extent is true, it requires the user to write a rather large amount of Haskell code in order for the tool to generate correct code for this type inference. This

means that a large part of the type checker still has to be written by hand because many non-trivial type rules will require access to the context or for types to be unified (this is a required process for Hindley-Milner type inference) for which code is not automatically generated.

2.2.4 Typical

The final tool we will be discussing is Typical[10]. Typical is a language that uses ML[6] as the base of its language. This means that all terms in Typical are typed and such as a proof of concept of Typical, the type checker for Typical is built using Typical. Typical also has some extra declarations that are not in ML, these are the ability to define name-spaces for the context and scoping rules. In the tools we have discusses so far the users have had to write such features of their type system. Typical also has in built functions that retrieve types from the context and adds new items to a context, which is again features that the previous tools did not have. Unlike our previous examples that use attribute grammars to perform their type checking, Typical uses the abstract syntax tree given by the parser. This means that Typical can be used with any parser generator that targets Typical's target language (Java), although it is stated that some modification may be required. The required modification has been performed on the Rats! parser generator[18]. To understand a type system written in Typical one must be familiar with ML or another language similar to it, such as Haskell, since the vast majority of the code in Typical is ML.

3 Design of the Notation

Before type checking code can be generated, a notation needs to be devised for representing a type system such that a program can read the given specification and generate the required code. The way the type system will be specified is through a domain specified language has been designed called *JET Expresses Types* (JET). The way type rules are written in JET are analogous to the natural deduction rules to define the type system. This this logical because other parts of the compiler pipeline already have domain specific languages that are used to generate code. JET is written using BNFC[19] for the parser, and the code generator is written using Haskell[20] with the ghc compiler[21].

In JET, rather than using the concrete syntax that is used in the natural deduction rules, the abstract syntax is provided by the user to define the type rules. The abstract syntax is used for a number of reasons. the first reason is that we are going to be type checking input that has already been parsed, therefore we are going to be type checking the abstract syntax tree and not the input of the concrete syntax. The second reason is that it is possible to type check an intermediate representation of a language. Intermediate representations may only be represented through abstract syntax and have no concrete syntax, for example, Haskell's intermediate representation (known as core) is type checked during Haskell's compilation process[21]. Type rules in JET have a similar form to the natural deduction rules, such that there is a rule name followed by some judgments and then the consequent. This provides a common way of representing type rules even if the way of writing the rules in JET uses an ASCII representation of the natural deduction rules.

The code being generated is based off of the type checking and type inference algorithm described in Ranta [1]. This is because the algorithms described are simple yet powerful enough to express common type rules, such as arithmetic expressions and also allow for the use of functions and variables. In addition it makes the code generation quite simple as there is a direct translation of type rules to a function in the algorithm. We will be generating Haskell code, since we can use the features in the Haskell language such as pattern matching and the use of monads to help produce relatively simple code from a given type specification.

3.0.1 Example Languages

When discussing the features of JET, they will be discussed in relation to two example languages. These languages fulfil the aims for what language features JET should be able to type check (given in chapter 1). The languages also exhibit different features features and paradigms so that we can demonstrate the diversity of languages that JET can type check.

The first of these languages is called Ad. The language is a small imperative block structured language similar to Ada. An Ad program is a series of type definitions followed by a procedure. Procedures have a list of variable declarations as parameters. Procedures consist of a list of declarations followed by a list of statements. Declarations can introduce new variables or subprograms (collective term for a function or procedure). A variable declaration consists of an identifier annotated with the type of the new variable, i.e. `x : Int`. A function differs from a procedure in that, a function returns a value whereas a procedure does not. There are a number of built-in statements in the language, consisting of control flow statements; a statement to introduce a new block; an assignment statement; a procedure call statement and a print statement. The language also has a small number of built-in expressions: literal, arithmetic, Boolean, variable, and function call expressions. Ad has two primitive types: integer and Boolean. It is also able to express more complex types such as record and array types. It can define type synonyms in Ad, type synonyms are similar to typedef's in C such that it is possible to give a more complex type a name to reduce repeated type definitions in variable declarations.

The second language is an implementation of the *Simply Typed Lambda Calculus* (STLC). STLC is a simple functional programming language. A program in STLC consists of a single expression. In the language there is access to simple arithmetic expressions; and more complicated expressions such as an if-expression and fix-expressions (this allows for recursive functions), lambda abstraction (this creates an anonymous function that takes an expression as a parameter and has a single expression as a body), and function application that applies a given function with a given parameter. The type system, while having the same primitive types as Ad, is quite different to that in Ad. The major difference is that it has a function type which is a type that takes an input to an output, often represented as $\tau \to \sigma$ where τ is the type of the parameter and σ is the type returned by the function. This also means that the type system in STLC treats functions as "first-class citizens" as one can give functions as parameters to other functions and return new functions from a given function. STLC is based off of the lambda calculus that is given in Pierce and Benjamin [22], they go into much more detail and have more extensions to STLC than will be considered here.

Full definitions of the abstract syntax and formal type systems for both of the example languages can be seen in Appendix B. JET specifications for both of these languages are also provided.

3.1 Overview of the JET Language

The JET language has two main structural components: some Haskell code in an initial section[1], followed by a list of type rules. The initial Haskell block is to allow the user to define functions and types that are to be used in the following type rules.

In order to write a type system in JET, the user will also have to provide a context type, which is an instance of a Haskell type class that defines a number of functions that will be useful when interacting with the context, such as looking up a variable in the context and adding an item to the context. This type class has been written in such a way to support as many kinds of contexts as possible and not limit what kinds of languages JET can support. Along with the type class, there is also an example of how to implement an instance of the type class using a Haskell Data Map[23] which the user may use or modify.

3.1.1 Initial Inline Haskell

The block of initial Haskell code, during code generation, will be placed just under the module definition but before the code for the type rules. The inline Haskell code blocks take the form of $\{Code\}$. It has a similar syntactic structure to code blocks in Flex and Bison (and their variants) so that the format will feel familiar to compiler writers. As previously mentioned, this is where a user can define functions and types required for the generated type checking code. For example this is where a user can define the type for the context along with defining the instance of the type classdefine , `JetContext`, for the users custom context type. It also allows for the ability to import any other Haskell modules, meaning that the user can import their Abstract Syntax Definition for that language that may be defined in a separate Haskell module. A complete example for a block structured imperative language, similar to Ad but with less features, can be seen in Listing A.1. Here the context is defined to be a 3-tuple which contains return type of the current subprogram, along with two name-spaces, one for functions; and one for variables. Namespaces, and their uses are discussed in detail section 3.4. These namespaces utilise the `JetContextMap`

[1] This is similar Flex and Bison's C code blocks to provide data structures and functions for the scanner and parser.

provided to the user. `JetContextMap` is a list of Maps, where each Map is a scope in the code. The instance of the type class `JetContext` for `JetContextMap` is already provided to the user.

Having this inline code allows the user to specify exactly how they wish the context to function. It also allows for a set of standard functions that the user can use to interact with the context, along with adding their own custom functions if and when they need them.

3.1.2 Type Rule Format Overview

The format of the type rules is extremely similar to the way type rules are represented in the natural deduction logic. This allows for simple translation of a natural deduction type rule into one of the type rules written in JET. We will delve into more detail about the format of the type rules, but the basic structure of the type rules is as follows:

⟨*Rule*⟩ ::= '`typerule`' ⟨*Identifier*⟩ '`<-`' ['`if`' ⟨*TypePremiseList*⟩ '`then`']
⟨*TypeConsequent*⟩ '`return`' ⟨*InlineHaskell*⟩

Figure 3.1: BNF Grammar rule for defining a JET type rule

The type premise list encased within the if-then is the list of judgments to prove the consequent. It is possible for a type rule to require no judgments to prove the consequent so the if-then is optional, therefore the type premise list can be forced to be nonempty. The type premise after the if-then is the consequent we are trying to prove. It then has a return keyword followed by some inline Haskell. This is to allow the user to return any data that is required for type checking future AST Nodes. For example, in our block structured language Ad, blocks are list of declarations followed by a list of statements. When checking the declarations it will be adding new variables and subprograms to the context. When it finishes checking the declarations it will return the context so that those variables and subprograms can be used in the statements. If the context was not returned then those declared variables and subprograms would not be found in the context when they should have been. The grammar of JET is designed to replicate, as much as possible, from the natural deduction rules because they are already an accepted way of formally representing type rules. The formal grammar can be seen in its entirety in Appendix A.

STLCIszero: $\dfrac{\Gamma \vdash e : Int}{\Gamma \vdash iszero\ e : Bool}$ STLCInt: $\dfrac{}{\Gamma \vdash n : Int}$

(a) Type rule for iszero expression in STLC

(b) Type rule for integer literal expression in STLC

AdEqT: $\dfrac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : Bool}$

(c) Type rule for equality expression in Ad

Figure 3.2: Examples of simple type rules

3.2 Type Rules

Type rules are How the type system, for which the user wishes to generate the type checking code, is expressed Therefore, the structure and semantics of type rules will determine what language features JET can express and successfully generate valid code for the type checker. We are going to be using the example languages seen in subsection 3.0.1 as motivation for the existence of features and their use.

3.2.1 Basic Type Rules

The type consequent is the most important part of the type rule because it tells us exactly what part of the abstract syntax is being type checked along with the type associated with that piece of abstract syntax. For example the type consequent for the type rule given in Figure 3.2a for STLC is:

```
(Expr Iszero e) : TBool
```

The given consequent represents exactly the same thing as the consequent in the natural deduction rule, it uses the abstract syntax that is have specified in Appendix B. it is worth to note that to associate a type with the consequent the consequent must be followed by a colon and the Haskell type constructor or variable of the type to associate with the consequent. In this case it is the type constructor Haskell that identifies the Boolean type which is `TBool`. To demonstrate how to write a type rule in JET, and the resulting generated code, the natural deduction rule given in Figure 3.2b will be used as an example. The type rule is very simple, having no premises or side conditions, meaning that the type rule in our language should be just as simple:

```
typerule STLCInt <- (Literal LitInt n) : TInt
   return {Succ ()};
```

There are three constituent parts to this type rule. The first (`typerule STLCInt`) has no semantic meaning and is instead just an identifier for the type rule, to be used for documentation purposes of the type system. The second is the consequent of the type rule (taking a form similar to the previously discussed example) using the abstract syntax to pattern match on the literal expression for integer. It is also annotated with the representation, in the abstract syntax, for integer in STLC. The final part is the return statement. This is where any extra data is returned if necessary so that is can be used in the rest of the type checking. Since this is just checking a literal integer there is no extra information that is useful from this type rule so it just returns an empty tuple. The type checking function JET generates has to assert than the expected type is the same as the type associated with out type rule. There are a number of way to do this in Haskell, the way in which JET does this is within an if-expression in Haskell:

```
1 checkLiteral (LitInt n) jetCheckType ctx = do
2    if jetCheckType == TInt then Succ ()
3    else Fail (makeCheckError (LitInt n) jetCheckType TInt)
```

The suffix of the name of the function is the same as the first identifier in the consequent. This is because the first identifier in the consequent must always be the type name of AST type that is being type checked. In this case it is checking a `Literal`, in other cases it could be checking an `Expr` in which case the first identifier in the consequent would be replaced. This allows JET to pattern match against any constructor of a given AST node. For example, if an arbitrary expression in our Ad language needed checking that expression could be a literal expression or an addition expression or any other expression within the language. If JET were to generate a unique function for each possible kind of expressions, i.e. `checkEAdd`, then it could not match against any arbitrary expression, the type checker would need to know the kind of expression before it can call a function. However, if each expression is within the same function, then JET can use Haskell's pattern matching features to identify what kind of expression it is and then execute the relevant type checking code for that kind of expression. From this the type consequent can be generalised so that it has the format of `HaskellType HaskellTypeConstructor`.

The first parameter (`LitInt n`) of the function is the Haskell type constructor specified in the consequent. This allows for the required pattern matching so that the type checker can call the correct type checking code. The second parameter (`jetCheckType`) is the expected type to be associated with this type rule. This is the parameter that the equality check will be performed on when asserting whether or not the actual and the expected type match. Finally, the last parameter is the current context which thr type checker is checking against. In this case the type rule does not interact with the context, however, more complicated type rules may do so, hence it is required to be a parameter to every type checking function so that is never lost when performing the type checking.

$$\text{AdNullT: } \frac{}{\Gamma \vdash \lfloor null \rfloor \ valid}$$

Figure 3.3: Type rule for a null statement in the Ad language

Since this type rule has no premises, the body of the function only has to check that the expected and actual type match. If they do then it succeeds and returns whatever the inline Haskell was in the return statement of the type rule. If not then it fails and generates some error message from a user defined error function.

How the type check function is generated has been discussed, however that is not the only function that is required. The infer function is used to return the type associated with a type rule so that the type can be used in other areas of the type checking. For example, in the type rule show in Figure 3.2c it has two type premises both with type τ, in this case τ is an arbitrary type but both instances of τ are the same. Therefore, a function is required so that the type associated with a piece of syntax can be retrieved (more commonly this is known as type inference[3]). Going back to the previous example (to see examples of type rules with premises go to subsection 3.2.3), code needs to be generated that will return the type associated with our type rule. In this case TInt needs to be returned, in this case the following code is produced:

```
inferLiteral (LitInt n) ctx = do Succ TInt
```

The function signature for the infer function is similar to the signature for the check function. The difference is that it no longer require the `jetCheckType` parameter as it is no longer checking the type associate with the type rule matches some expected type, instead it returns the type associated with our type rule. Therefore, the function does not require the if-expression, instead it just needs to return the type `TInt`.

3.2.2 Typeless Type Rules

Some languages require that every type rule must have a type associated with it, STLC is an example of such a language. However, this is not the case for all languages. For example, in the Ad language, only expressions have types associated with them, every other syntactic part of the language is simply referred to as $valid$ if all of the premises of the type rule are true. The simplest of these type rules is the null statement in the Ad language. Seen in Figure 3.3, this is a simple type rule where if it is successfully parsed it will always be true as it has no premises. This type rule given in JET is represented as:

```
typerule AdNullT <- (Stmnt SNull) return {Succ ()}
```

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$$

(a) Type rule for STLC if-expression

$$\frac{\dfrac{}{\Gamma \vdash 0 : Int}\,\text{Int}}{\Gamma \vdash iszero\ 0 : Bool}\,\text{Iszero} \qquad \frac{\dfrac{}{\Gamma \vdash 0 : Int}\,\text{Int}}{\Gamma \vdash pred\ 0 : Int}\,\text{Pred} \qquad \frac{\dfrac{\dfrac{}{\Gamma \vdash 0 : Int}\,\text{Int}}{\Gamma \vdash pred\ 0 : Int}\,\text{Pred}}{\Gamma \vdash pred(pred\ 0) : Int}\,\text{Pred}}{\Gamma \vdash if\ iszero\ 0\ then\ pred\ 0\ else\ pred\ (pred\ 0)}$$

(b) Proof tree example for a STLC expression (Type rule names have been abbreviated)

Figure 3.4: Type rule for if-expression and corresponding proof tree example

Here, unlike in previous type rules tht have been discussed in subsection 3.2.1, no type is associated with the type rules at all. This represents the concept of marking something as $valid$ in the natural deduction rules. The code that is generated also substantially different (and much simpler) as it no longer needs to check the type as there is no type. It also no longer needs to generate an infer function to return a type as again there is no type to return:

```
checkStmnt SNull ctx = do Succ ()
```

You may also notice that the parameter `jetCheckType` has also been omitted as there is no need for it since there is no type to check.

The same functionality could be aschieved without this feature by having an internal type that represents $valid$. In the Ad language `TNNone` could be used for this purpose giving us the type rule:

```
typerule AdNullT <- (Stmnt SNull) :  TNNone return Succ ()
```

However, this is redundant because it generates more code than necessary as it now has a type to check, whereas before it did not. This could potentially lead to a loss in performance since it is now having to perform an equality check and also has to give the `jetCheckType` parameter to the check function. Additionally it will generate an infer function since there is now the possibility to infer the type of such a statement whereas in our original type rule it did not have this potential before.

3.2.3 Type Premises

So far we have only discussed in detail type rules without premises. However, to be able to type check and traverse the whole abstract syntax tree

given to JET by a parser, it needs to generate code that uses premises. For example, if given the natural deduction rule given in Figure 3.4a, the type rule in JET will need to represent the premises given in this type rule. The code JET generates will also want to call the code that checks all of the premises given for the type rule. Considering the code snipped for STLC, `if iszero 0 then pred 0 else pred (pred 0)`, its proof tree can be seen in Figure 3.4b. The method of proving a sentence in a language is through recursive application of type rules until it reaches a type rule that has no premises causing the proof to terminate. In this case the literal integer type rule acts as the base case for all of the premises. The type rules that have already discussed are able to act as base cases to the recursive nature of type checking that we wish to generate code for. Therefore, the code we wish to generate for type premises will need to replicate this recursive nature we desire.

Type premises have a similar syntax to type consequents. More information that is required for a type premise. Namely the context which the type premise is meant to be proved with respect to. For now we will be looking at type rules that do not interact with the context (go to section 3.4 to see type rules that interact with the context). This gives the general form of a type premise to be:

⟨*TypePremise*⟩ ::= ⟨*InlineHaskell*⟩ '|-' '(' ⟨*Identifier*⟩ ⟨*Identifier*⟩ ')' [':' ⟨*Type*⟩]

Here, the first $Identifier$ is the type of the AST node for the premise; the second $Identifier$ is the variable that the premise is meant to check. : $Type$ is the optional type that is associated with the type premise. We are now in a position to write a JET type rule for our type rule given in Figure 3.4a:

```
1 typerule IfExpr <- if {ctx} |- (Expr e1) : TBool,
2       {ctx} |- (Expr e2) : t,
3       {ctx} |- (Expr e3) : t then
4    (Expr If e1 e2 e3) : t return {Succ ()}
```

A type premise list in JET is a comma separated sequence of type premises. This is the first occurrence of a type variable being used in a type rule. There are a few sematic rules about type variables. If the user has said that the type of a premise is a type variable, and this is the first occurrence of the type variable then JET will generate a call to an infer function to get the value that the type variable is to represent. Again, if the user haskell said that the type of a premise is a type variable, but this time JET has seen the type variable then it will generate a check function where the type to check is the already defined type variable. This is to assert the two type variables are consistent. Finally, if the user uses a type variable in the consequent then the variable should have been defined somewhere in the type premises. From this set of rule JET ends up generating the following code:

ArrAcc: $\dfrac{\Gamma \vdash e : Arr\ \tau \qquad \Gamma \vdash n : Int}{\Gamma \vdash e[n] : \tau}$

(a) Type rule for array access in Ad

Fix: $\dfrac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash fix\ e : \tau}$

(b) Type rule for fix point operator in STLC

Figure 3.5: Type rules with type variables as part of type constructors

```
1  checkExpr (If e1 e2 e3) jetCheckType ctx = do
2     var1 <- checkExpr e1 TBool ctx
3     t <- inferExpr e2 ctx
4     var2 <- checkExpr e3 ctx
5     if jetCheckType == t then Succ () else Fail
          (makeCheckError (If e1 e2 e3) jetCheckType t)
6  inferExpr (If e1 e2 e3) jetCheckType ctx = do
7     var1 <- checkExpr e1 TBool ctx
8     t <- inferExpr e2 ctx
9     var2 <- checkExpr e3 ctx
10    Succ t
```

As can be seen above, the final line of both functions are similar to the bodies of functions that we have seen that did not have type premises. The other key point to note, is that the code generated for the type premises appears in both of our generated functions. This is because, in both instances, it needs to prove premises otherwise anything we state about the consequent could be incorrect. Premises are evaluated in the order of left to right in the premise list, and the return value from check functions is stored in a variable with name var_n where n is the next free natural number such that the variable name is unique.

Type Variables as Parameters to Type Constructors

Jet can also use type variables as parameters to the constructor of a full type definition. Examples of these types are array types in the Ad language, and a function type for STLC. All elements in an array, in Ad, are all of the same type. Therefore the type rule for accessing an element of the array will need to extract the type of the elements of the array. In the natural deduction rules this is trivial, you represent arrays as $Arr\ \tau$ where τ is the type of the elements of the array. We can use a similar form of representing such types in JET. For example, the type rule for array access in Ad is seen in Figure 3.5a. We can write the first type premise in JET in almost exactly the same way:

```
{ctx} |- (Expr e) : TArr t
```

From this JET generates code of the form:

```
TArr t <- inferExpr e ctx
```

This code asserts that the type returned from the infer function must be an array type, the generator then gets the parameter of the array type and stores it in some variable t. The generator will also have to add the variable t to the list of known type variables so that if another premise required the use of this type variable then we can generate a check function instead of an infer function. There is one issue with the code that we generate at the moment. If the return of the infer function does not match the pattern function then the type checker will error, but not with the defined error function since the error was an internal Haskell error. Instead the right hand side of the binding will want to call our error function in case of a bad pattern match, from this criteria the generator produces the follow code:

```
TArr t <- case inferExpr e ctx of Succ jet0@(TArr t) -> Succ
    jet0; Succ t -> Fail (makeInferError t t); x -> x
```

Side Conditions as Premises'

There is another example of such a type rule, but instead as part of STLC, that can be seen in Figure 3.5b. There is the type $\tau \rightarrow \tau$ where both instances of τ are the same. We would like to write the type premise as:

```
{ctx} |- (Expr e) : TFun t t
```

Where this type premise asserts that both instances of `t` are the same. However the code that is generated for this is not correct as it does not assert that both instances of `t` are the same, instead `t` is always the value of the second parameter to the type constructor. Therefore, the type premise is written as:

```
{ctx} |- (Expr e) : TFun t1 t2
```

A side condition premise is also written such that it asserts that `t1` and `t2` are the same. A side condition premise is piece of inline Haskell that is inserted into the generated code as a monadic statement The use of the side conditions can be extremely varied in functionality and can do anything the user requires that cannot be done as part of normal type premises. One of the major uses of Haskell side conditions is for retrieving or expanding the context that we will see in section 3.4. From this the type rules is written as follows:

```
1 typerule Fix <- if {ctx} |- (Expr e) : TFun t1 t2,
2       {if t1 == t2 then Succ t1 else Fail "type error"} then
3    (Expr Fix e) : t1 return {Succ ()};
```

3.3 List Notation Type Rules

Many languages, in their abstract syntax, have nodes that take a list of nodes to be one of their children (an example of this can be seen as a

sequence of statements that make up the body of a procedure or function as part of the Ad language in Appendix B). In the set of features discussed, there is no simple way to represent type rules that act on these kinds of nodes.

3.3.1 List Notation in Type Premises

Using list notation in type premises is similar to how normal type premises are used. The difference is, instead of encasing the premise with parentheses, it is instead encased within square brackets. This so that it matches how lists are defined in Haskell, therefore, the user should be familiar with square brackets being associated with lists. The example for the list of statements is given as `{ctx} |- [Stmnt stmnts]`. This is exactly the same as a normal type premise, with only the type of brackets used different, so that both the user and the tool can tell the difference. The code that is produced from this type of premise is also very similar to the code produced from a normal premise: `var1 <- checkStmntList stmnts ctx`. How the parameters are passed, along with how the result of the premise is bound to a variable, is exactly the same. The only difference being is the function name has `List` as a suffix. If we did not have this suffix then we would call `checkStmnt` which would lead to a type error as `checkStmnt` has the type: `Stmnt -> Context -> JetError a`, whereas the function we are trying to call would have to have type: `[Stmnt] -> Context -> JetError a`. The difference being that, one expects a statement whilst the other expects a list of statements.

3.3.2 List Notation Type Rules

We have discussed how list notation is used in premises. However, that is not useful unless code can be generated for type rules using list notation. Type rules for list notation bear a resemblance to normal type rules. They have different semantics for the consequent but the type premises have the same semantics we have previously discussed. Similarly to premises, instead of being encased in parentheses, they are encased in square brackets. The major differences occur in the variables that are defined within the rule and what semantic meaning they have. The simplest form is to just define the node type and nothing else, for example: `[Stmnt]`. This will pattern match on the empty list, in this case when all statements have been successfully type checked. The next pattern has the node type and a variable name, given as: `[Stmnt s]`. This pattern will match on the singleton list, where the element within the list is given the variable name in the type rule. The final pattern that exists is the list constructor pattern, this is given as the node type but is now followed by two variable names. The

$$\text{StmntList: } \frac{\Gamma \vdash \lfloor stmnt \rfloor \ valid \qquad \Gamma \vdash \lfloor stmnts \rfloor \ valid}{\Gamma \vdash \lfloor stmnt;\ stmnts \rfloor \ valid}$$

Figure 3.6: Type rule for a sequence of statements in the Ad language

first variable represents the head of the list to be type checked (the first item), the second variable represents the tail of the list to be type checked (The rest of the list after the head has been removed). The variables in these type rules can be used the in the premises, the same as any other variables.

A example of how to define a type rule in list notation is when type checking a list of statements. The type rule in natural deduction logic can be seen in Figure 3.6. The premises in this type rule make a recursive type rule, and the type rule in JET will have to reflect this meaning. In the scenario given, the list statements must non-empty, we can create type rules that assert that the non-empty property must hold.

Listing 3.1: Type rule in JET for a seqeunce of statements

```
1 typerule StmntListCons <- if {ctx} |- (Stmnt stmnt),
2       {ctx} |- [Stmnt stmnts] then
3    [Stmnt stmnt stmnts] return {Succ ()};
4 typerule StmntListSngltn <- if {ctx} |- (Stmnt stmnt),
5    then [Stmnt stmnt] return {Succ ()};
```

We therefore have the type rule in JET that can be seen in Listing 3.1. As demonstrated to create the behaviour that is desired, two type rules are needed, one for when the list is of arbitrary length, and another for when the list contains a single item. This structure forces the list to be nonempty and the type checker will error if the list happens to be empty. Another important aspect to note is that we can recursively call the same type rule in order to type check the while list. This can be seen in the list constructor type rule, where the current statement is type checked but then also go on to type check the rest of the list. These patterns should be able to handle any usual type rule that a user is going to write. However, they do not cover every possible pattern that could occur as they do not replicate how Haskell does pattern matching on lists but instead uses a subset of the patterns that Haskell has available. It could be possible, in future, to expand these patterns so that they do match all of the patterns that Haskell can handle however, we did not require this feature to type check the example languages given.

3.4 Interacting with the Context

Using side conditions to update and lookup items within the context is very common when specifying the natural deduction rules. We have discussed in subsubsection 3.2.3 how side conditions work in JET, and how the code is generated for them. To make interacting with the context, through Haskell side conditions, easier and more consistent, there is a Haskell type class that has been provided to the user to give them a common interface to interact with the context[2]. It is important to note that these modules do not define the context for the user; the user will have to define their own context type and instantiate the Haskell type class for their own context type. This is because many different languages can have contexts that work differently and letting the user define their own context gives them greater control on the semantics of the context. There is an example provided to the user of how to do this using Haskell Data Maps[23]. The name of the type for this example is `JetContextMap`. The semantics of the each function will be explained using the provided example.

3.4.1 Empty Contexts and Blocks

Listing 3.2: Example of declarations in a block structured language such as Ad

```
1 declare
2    x : int; -- This will succeed
3 begin
4    declare
5       x : bool; -- This will also succeed because the
            variable is re-declared in a new block
6       x : int; -- This will fail because the variable was
            re-declared in the same block as the previous
            declaration
7    begin
8    end;
9 end;
```

The basic thing you can do with a context is create an empty context or add a new block to the context. An empty context should do exactly what the name suggests, it should be a new context that has no items in it. Since the user implements all the functions they should ensure that their function meets these semantic requirements. There is an assumption that all languages will have a block structure even if that block structure is trivial, and if a language is not a block structured language it can be modelled by having a context which will only ever have one block. How the context represents blocks is up to the user. However in our `JetContextMap` example a single block is a map and our context is made up of a list of

[2]The definition of this type class (called `JetContext`) is provided in Appendix C

$$\frac{\langle\emptyset\rangle^\frown\Gamma \vdash \lfloor decls \rfloor valid \qquad \langle\{decls\}\rangle^\frown\Gamma \vdash \lfloor stmnts \rfloor\ valid}{\Gamma \vdash \lfloor\ \text{declare}\ decls\ \text{begin}\ stmnts\ \text{end}\ \rfloor\ valid}$$

Figure 3.7: Type rule for a new block in the Ad language

blocks. The reason for this is simple; consider the Ad program given in Listing 3.2. We want to be able to re-declare variables as long as the variable to re-declare does not exist in the current block. If the context was just a single map then there would be no way of knowing whether or not a variable was declared in the current block; we would only be able to know if a variable was declared in any block. This gives us the formal definition of the context to be a sequence of functions which maps names to types given as $\langle Name \rightarrow Type \rangle$

These functions can be used anywhere inline Haskell can be used. However, they will most likely be used in the context definition for type premise's. An example of natural deduction rule defining a new block for the Ad language can be seen in Figure 3.7. The first premise states, given a sequence containing an empty set (conceptual a mapping of names to types containing nothing), concatenated with our current context, check our declarations. The second premise states check the statements of the block given a new block containing all of our declarations concatenated with the current context. The following type rule in JET represents this natural deduction rule:

```
1 typerule BlockT <- if {(newBlock ctx)} |- [Decl decls],
2       {var1} |- [Stmnt stmnts] then
3    (Stmnt SBlock decls stmnts) return {Succ ()};
```

`var1` is used for the context in the second premise because `var1` is the context returned from the first premise which has all of the declarations added to the context. This is required so that those declarations are visible to the statements in the block.

3.4.2 Lookup and Expand

The most common use of the context is when attempting to retrieve types of objects from it. We will want to do two main actions with variables, the first is to retrieve the type of a variable from the context; the second is to add a new variable with a given type to the context. Retrieving the type of a variable from the context is given by the natural deduction rule in Figure 3.8a. The consequent is trivial and is the same as the ones we have seen throughout subsection 3.2.1. We still need to retrieve the arbitrary type τ from the context. This can be done by using the function, given by the `JetContext` type class, `lookupContext`. This function, given

$$\text{VarT: } \frac{}{\Gamma \vdash v : \tau}\,[\tau = lookup(v, \Gamma)]$$

(a) Type rule for using a variable in the Simply Typed Lambda Calculus and Ad languages

$$\text{AdVarDecl: } \frac{(head(\Gamma) \oplus v : \tau)^\frown(tail(\Gamma)) \vdash \lfloor decl \rfloor}{\Gamma \vdash \lfloor v : \tau; decls \rfloor\ valid}\,[v \notin dom((head(\Gamma)))]$$

(b) Type rule for variable declaration in the Ad language

$$\text{AdVarDecl: } \frac{expandContext(v, \tau, \Gamma) \vdash \lfloor decls \rfloor}{\Gamma \vdash \lfloor v : \tau; decls \rfloor\ valid}$$

(c) Type rule for variable declaration in the Ad language using functions from the Haskell type class

Figure 3.8

some a parameter to identify the item in the context (such as the variable name), will return the type associated with the input, or will return some fail message within the `JetError` monad. When using this in a Haskell side condition we will want to bind the resulting type from the context to the variable that we are going to use as the type for the consequent. If we say that the return variable of the consequent is `t`, then we will want to write the side condition as `{t <- lookupContext var ctx}`, where `var` is the variable to lookup and `ctx` is the given context to perform the lookup in. When generating the code from a Haskell side condition, it is simply put directly into the generated code in the position it occurs in the premise list (In this example it is the only premise so that does not matter just as long as it appears before the return monadic statement).

Now types can be retrieved from the context. However, that is not useful unless objects can be added to the context. We will be using the example that can be seen in Figure 3.8b, taken from the Ad language. The type rule has a very complicated predicates for the new context and also for the side condition for the type rule. However, the functionality can be encoded into the `expandContext` function. The `expandContext` functions takes as parameters the object, the type to associate with the object and the context to add the object to. The function will return a monad which can contain either the new context or an error, where if successful it will return the new context otherwise it will return an error. If we were to then use this function to write the natural deduction rule again we would get the rule given in Figure 3.8c.

The new version of the natural deduction rule is now much easier to write in JET because we can use the function from our type class. We write the

$$\frac{\Gamma \vdash params : \tau_{params}}{\Gamma \vdash f(params) : \tau}\,[\tau = lookup((f, \tau_{params}), F)]$$

Figure 3.9: Type rule for function call in Ad

type rule in JET as the following:

```
typerule VDeclT <- (VarDecl VDecl var t) return
   {expandContext var t ctx};
```

There is no need for the premise in our JET type rules as we write that using the list notations. This type rule is simply to add the variable to the context therefore we need to return the new context which has been expanded with the variable rather than returning the Unit type that we have seen in every other type rule so far. The code we produce for this is similar to the typeless type rules seen in subsection 3.2.2, but with the expand function in place of `Succ ()`:

```
checkVarDecl (VDecl var t) ctx = do
   expandContext var t ctx
```

3.4.3 Name-spaces

Name-spaces are not directly supported in the JET language, as they are in Typical[10]. They are a common construct in programming languages and a useful way of splitting the context into multiple parts. This means that in JET, we can have a variable, a function, a procedure and a type synonym with the same name but we can figure out which one we mean based upon when it is being used. For example, in the Ad language we will want to have multiple name-spaces; one for variables, one for functions, one for procedures and a final one for type definitions. Whereas, in the STLC, we only have one name-space for variables. For our Ad language we need to construct our context in such a way that we can have all of the name-spaces we require. We do this using a 4-tuple where the items for variables, functions and procedures are the block structures we discussed in subsection 3.4.1. Whereas our structure for type definitions is just a map because types can only be defined at the start of the program. The formal definition for the current context we require for Ad is (V, P, F, T) where V is the variables, P is the procedures, F is the functions and T are the type definitions. However Γ will still be used to denote the whole context. Therefore, while trying to type check a function call, we will get the natural deduction rule given in Figure 3.9. If we are using the context definition for Ad (in Appendix B) then we would be able to write this type rule in JET, using list notation to get the types of the parameters, as:

```
1 typerule FCallT <- if {ctx} |- [Expr params] : tparams,
2       {t <- lookupContext (Func ident tparams) ctx} then
```

```
3     (Expr ECall ident params) : t return {Succ ()};
```

The Haskell side condition used here is very similar to the one that was shown previously when describing how the function `lookupContext` works. However, this time the first parameter has a different form. In this case it has the form of the name-space which we want to look for the object in. This is followed by the object that we wish to find, here the object is the identifier of the function and the types of the parameters to the function. Therefore, `ident tparams` is analogous to (f, τ_{params}) in the natural deduction rule. So we give the whole context to the function and we let the function decide what name-space to look in based upon the pattern of the first argument (i.e. what constructor is used for the first argument) in this case it is the `Func`, from the type:

```
data ContextKey = Ret | Proc Ident [Type] | Func Ident
    [Type] | Var Ident | Typ Ident
   deriving (Eq, Ord, Show, Read)
```

This is the type that is used to access the context. The type here has one more item in the tuple that we didn't initially give in our formal definition of the Ad context, that is the constructor `Ret`. That item is used to denote the current return type of the current subprogram we are in. This means the new formal definition for the context for the Ad language is a 5-tuple given as (τ_{ret}, V, P, F, T) where the variables that existed previously retain their definition and τ_{ret} denotes the current return type. This piece of information is required so that return statement can be correctly type checked when it is used. This can be used by the `lookupContext` and `expandContext` functions, just like the other constructors for that type, the only difference is the constructor does not require any parameters this means we can write the type rule for the return statement to be:

```
1  typerule ReturnT <- if {ctx} |- (Expr e) : t,
2        {tret <- lookupContext Ret ctx},
3        {if t == tret then Succ () else Fail "Type error"}
4     then (Stmnt SReturn e) return {Succ ()}
```

As can be seen, the type for the expression to return is inferred and then there is an if-expression to assert that the type of the expression is the same as the current return type.

4 Evaluation and Conclusion

Given all of the features and type rule patterns we have discussed in chapter 3 we can successfully type check both of our example languages. As per the aims given in chapter 1, we have discussed how to type check all but user defined types. The full type specification in JET, along with the natural deduction rules that we are implementing in our JET specification, can be seen in Appendix B. Whilst we have not discussed how to type check user defined types and record types, the JET specification for the Ad language does support these language features and does successfully type check them. We did not discuss these type rules because there are no other features in the language that allow for a better representation of user defined types and record types than the representation that can be seen in the JET specification of the Ad language.

Both of our example languages have differing paradigms and language structure, meeting one of our first main criteria of the project which was to be able type check simple programming languages from different paradigms. The first of our languages, Ad was a small imperative block structured language which had variable, function and procedures along with a few primitive statements and expressions. Whereas STLC was a functional programming language that allowed for the use of functions as first class citizens, along with higher order functions. These two languages represent two main paradigms in programming languages and provide a base from which to implement other features in the future that would allow the extension of the type systems in these languages. The type system features that could be added to these languages are, in brief, for Ad, the use of extending record type to have the ability of subtyping and also implementing full object orientations; and in STLC, the addition of recursive types[3], [22] and polymorphic types[24]. Whilst these features are not a requirement, they were mentioned as possible end goals that a full tool (JET is currently still a prototype) would need to support. They would almost certainly be required for a complete tool to support a large variety of complex language.

We have discussed briefly whether JET met our criteria for success. However, now we will discuss in detail each required feature in our list of aims given in chapter 1. The notation is simple, there are a few more advanced features for type rules, but, the type rules are similar to if statements in general purpose programming language and should therefore feel familiar to programmers. In its current state the notation is not as easy to understand

or use as it could be. In order to type check an imperative language that has a relatively complicated context type associated with it the user may have to write a large amount of inline Haskell to get the generated code to function as required. This can make the given JET specification hard to read in places, an example of this is when trying to type check user defined types in Ad. We have demonstrated that we can generate code for simple imperative and functional programming languages, I would, therefore, state we have succeeded in type checking languages of varying paradigms. We have also succeeded in generating code for simple arithmetic expressions and simple statements as we can type check Ad which has both of these language constructs. Both of our example languages have variables, and we can type check both functions and procedures in Ad. Ad has complex types as part of its language and we can successfully type check these. The slight exception is that in order to type check record types correctly we do require a small amount of inline Haskell, however we can type check array types perfectly. STLC has higher order functions since it is a variation on the Simply Typed Lambda Calculus. Therefore we can successfully type check function types and lambda expressions.

I conclude that our notation (JET) has been a success as we can type check two complete simple languages. The notation is in most places. fairly simple. It's only in more complicated language constructs that the JET begins to lose its simplicity. Finally, we can type check all of our required language features and, with some work, we can type check user types, although not as succinctly as we would like. However, the further work we have mentioned would resolve many of the issues that JET currently has, therefore, making the tool far more usable.

We have very few ethical considerations to make. We have not used any people or animals as part of the testing process for the tool. There have been no experiments performed during the development of the tool or in any of its subsequent testing. Any software that was used during the development of the tool or used as part of the discussing during this report has been credited to the relevant people such as the use of BNFC and Haskell, we have not claimed any of the software as our own.

4.1 Further work

4.1.1 Generated Code Structure

Whilst we are able to generate correct code that meets our objectives, we are not generating particularly efficient code. Since JET is still a prototype tool the efficiency of the code generated is not a concern at this time. However, that is not to say that we should completely ignore the idea of

generating more efficient code. The biggest issue with efficiency is the order of generated code in our generated check functions with a given type. For example, the code we generate for our if-expressions in STLC is:

```
1 checkExpr (If e1 e2 e3) jetCheckType ctx = do
2    var1 <- checkExpr e1 TBool ctx
3    t <- inferExpr e2 ctx
4    var2 <- checkExpr e3 t ctx
5    if jetCheckType == t then Succ () else Fail "Type error"
```

In the event that our type `t` is not the same the type we wish to check, `jetCheckType`, then we will end up evaluating all the way to the if condition. This is not the most efficient code we can generate because we know `jetCheckType` is going to have to be the same as `t`, therefore we can say `jetCheckType` is `t` and turn the `inferExpr` into a `checkExpr`, so we would instead be generating the following code:

```
1 checkExpr (If e1 e2 e3) t ctx = do
2    var1 <- checkExpr e1 TBool ctx
3    var2 <- checkExpr e2 t ctx
4    var3 <- checkExpr e3 t ctx
5    Succ ()
```

In this case we will fail as soon as possible if `e2` does not have the correct type. Whereas before, we would not fail until after everything else had been evaluated.

4.1.2 Language Structure and Extensions

Syntactically, the language is similar to natural deduction rules, which sets it apart from a tool such as Typical[10]. This is the main advantage of this tool. It is designed to represent the natural deduction rules that it seeks to implement, however, it also allows the user to write their own Haskell to add functionality that, otherwise, JET would not be able to support.

One of the major short comings of JET is the representation of the context, both how it is defined and used throughout a given type system. As mentioned in subsection 2.2.4, I think the representation of the context in Typical is one of the best. It gives the user the ability to customise the type system as they wish but also provides a notation that is intuitive and easy to use. We tried, as much as possible, to make the definition of the type classes as easy to use as possible and to help give the user an idea of what information may be required to build a specification in JET. However, a better notation for defining namespaces would most likely make the tool easier to use than it is in its current form. Currently you have to write a fair amount of Haskell in order to get the context to act as required. Whereas, the notation given in Grimm, Harris and Le [10] is a much nicer way of representing namespaces. The inclusion of the predefined instance of the type class for `JetContextMap` does help provide a base for how the user

⟨*TypePremise*⟩ ::= ...
  | ⟨*Identifier*⟩ 'lookup' ⟨*Identifier*⟩ ':' ⟨*Identifier*⟩

Figure 4.1: Possible grammar rule for an extension to JET type premise's

will define their namespaces. The lack of language support for functions defined by the Haskell type class reduces the usability of the tool. This could be fixed by adding extra statements in the language. For example, if we were to attempt to lookup an item in the context in STLC, instead of writing the premise using inline haskell as `{t <- lookupContext var ctx}`, we could instead have a piece of syntax in JET that encodes this meaning. For example we could have a grammar rule that has a keyword lookup and takes an item to lookup and a context to lookup within (an example BNF grammar rule can be seen in Figure 4.1). Using this grammar rule the type premise would be rewritten as `var lookup context : t`. This would then be generated into the code that appeared in the initial type premise.

I have mentioned that we can type check user defined types. This is true, but, it requires excessive use of user defined Haskell functions and also takes advantages of how the language is structured and how we generate the output code. Using odd side effects of the language design to get the required behaviour obviously leads to unintuitive and hard to read type rules. Therefore, an area of future work to be performed, for JET specifically, would be to design and implement a notation that provides a more intuitive method of performing type checking for user defined types along with record types etc. This would greatly improve the usability of the tool, making it easier to implement a type checker for a language with a type system involving user defined types. It ay possibly make it quite simple to type check object oriented language which is class of language that we have not discussed but in the current state would require similar type rules to the ones in Ad for record types.

The type inference algorithm that JET generates is very simple, and is used to retrieve the type of expressions within a given language. Therefore, JET has no built-in support for type reconstruction or unification[3], [22]. This because there is no built-in way of specifying constraint typing relation[22]. This would require a notation in JET that allowed for the definition and use of type constraints. We will then be required to generate some type unification algorithm such that we can find a valid substitution that satisfies the constraints given. If we were able to add these features, then we would be able to start type checking languages of a similar class to standard ML[6], which as of at the moment would either require vast amounts of inline Haskell or cannot be done with JET.

4.1.3 Operational Semantics and Big Step Notation

The notation for the semantics of the evaluation and code generation of languages is very similar to the notation for type systems. They use the same natural deduction rules, instead of expressing types, it expresses evaluation or the resulting code. Pierce and Benjamin [22] use natural deduction rules to describe the evaluation of their Simply Typed Lambda Calculus. It is foreseeable then, since we have shown we can generate type checking code, that one could generate code for rules that express the operational semantics of a language.

4.2 Conclusion

It has been shown that it is possible to, at least partially, generate code for type checkers. It has also been show that code can be generated for languages of different programming paradigms, such as imperative and functional programming languages. For simpler languages generating type checking code is some what trivial. However, as the languages gain complexity, generating code becomes more difficult due to the corner cases of the languages semantics. This is especially prevalent in languages that have complicated structures for their context. We briefly began to touch on this issue in Ad, however not as much as we would if we attempted to type check languages such as Ada or C++. This is due to the nature of such languages, their type systems have grown in complexity and do not have form definition for their semantics and as such are open to interpretation. Therefore, more work will be required, to both extend JET with new features to support more complicated type systems (the ones discussed in section 4.1 are but a few), and also produce and test type checkers for more complicated languages using JET with possible extensions added to make this possible. It may be that some type systems will be able to be generated completely due to the "ad-hoc" nature of compilers and because many languages lack formal specifications for their type systems (along with other parts of the compiler). There is also the possibility of generating code for evaluation and code generation for languages which could make process of producing whole compilers and interprets much easier and quicker.

Bibliography

[1] A. Ranta, *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.

[2] J. Backus, 'The history of fortran i, ii, and iii,' *SIGPLAN Not.*, vol. 13, no. 8, pp. 165–180, Aug. 1978, ISSN: 0362-1340. DOI: 10.1145/960118.808380. [Online]. Available: http://doi.acm.org/10.1145/960118.808380.

[3] L. Cardelli, 'Type systems,' in *The Computer Science and Engineering Handbook*, A. B. Tucker Jr., Ed., Boca Raton, FL, USA: CRC Press, Inc., 1997, ch. 103.

[4] B. W. Kernighan and D. M. Ritchie, *The C programming language*. 2006.

[5] R. Milner, 'A theory of type polymorphism in programming,' *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978, ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[6] R. Milner, M. Tofte, R. Harper and D. MacQueen, *The definition of standard ML: revised*. MIT press, 1997.

[7] J. Bentley, 'Programming pearls: Little languages,' *Commun. ACM*, vol. 29, no. 8, pp. 711–721, Aug. 1986, ISSN: 0001-0782. DOI: 10.1145/6424.315691. [Online]. Available: http://doi.acm.org/10.1145/6424.315691.

[8] A. Van Deursen, P. Klint and J. Visser, 'Domain-specific languages: An annotated bibliography,' *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[9] J. Levine and L. John, *Flex & Bison*, 1st. O'Reilly Media, Inc., 2009, ISBN: 0596155972, 9780596155971.

[10] R. Grimm, L. Harris and A. Le, 'Typical: Taking the tedium out of typing,' *Presentation at the IBM Programming Languages Day*, 2007.

[11] A. Dijkstra and S. D. Swierstra, 'Ruler: Programming type rules,' in *International Symposium on Functional and Logic Programming*, Springer, 2006, pp. 30–46.

[12] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane and W. M. Waite, 'Eli: A complete, flexible compiler construction system,' *Commun. ACM*, vol. 35, no. 2, pp. 121–130, Feb. 1992, ISSN: 0001-0782. DOI: 10.1145/129630.129637. [Online]. Available: http://doi.acm.org/10.1145/129630.129637.

[13] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue *et al.*, *Ariane 5 flight 501 failure report by the inquiry board*, 1996.

[14] J. W. Backus, F. L. Bauer, J. Green *et al.*, 'Report on the algorithmic language algol 60,' *Numerische Mathematik*, vol. 2, no. 1, pp. 106–136, Dec. 1960, ISSN: 0945-3245. DOI: 10.1007/BF01386216. [Online]. Available: https://doi.org/10.1007/BF01386216.

[15] A. V. Aho, *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[16] D. Prawitz, *Natural deduction : a proof-theoretical study*. Mineola, N.Y: Dover Publications, 2006, ISBN: 0486446557.

[17] D. E. Knuth, 'Semantics of context-free languages,' *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, Jun. 1968, ISSN: 1433-0490. DOI: 10.1007/BF01692511. [Online]. Available: https://doi.org/10.1007/BF01692511.

[18] R. Grimm, 'Better extensibility through modular syntax,' *SIGPLAN Not.*, vol. 41, no. 6, pp. 38–51, Jun. 2006, ISSN: 0362-1340. DOI: 10.1145/1133255.1133987. [Online]. Available: http://doi.acm.org/10.1145/1133255.1133987.

[19] M. Pellauer, M. Forsberg and A. Ranta, 'Bnf converter multilingual front-end generation from labelled bnf grammars,' 2004.

[20] S. Marlow *et al.*, 'Haskell 2010 language report,' *Available online http://www. haskell. org/(May 2011)*, 2010.

[21] S. Marlow, S. P. Jones *et al.*, *The glasgow haskell compiler*, 2004.

[22] B. C. Pierce and C. Benjamin, *Types and programming languages*. MIT press, 2002.

[23] D. Leijen and A. Palamarchuk, *Haskell containers: Data.Map*, version 0.6.0.1, 26th Apr. 2019. [Online]. Available: https://hackage.haskell.org/package/containers-0.6.0.1/docs/Data-Map.html.

[24] L. Cardelli and P. Wegner, 'On understanding types, data abstraction, and polymorphism,' *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, Dec. 1985, ISSN: 0360-0300. DOI: 10.1145/6041.6042. [Online]. Available: http://doi.acm.org/10.1145/6041.6042.

Appendices

A JET Language

A.1 Grammar

⟨*TypeSystem*⟩ ::= ⟨*InlineHaskell*⟩ ⟨*TypeRuleList*⟩

⟨*InlineHaskell*⟩ ::= '`{`' ⟨*Code*⟩ '`}`'

⟨*TypeRuleList*⟩ ::= ⟨*TypeRule*⟩
| ⟨*TypeRule*⟩ '`;`' ⟨*TypeRuleList*⟩

⟨*TypeRuleList*⟩ ::= '`typerule`' ⟨*Identifier*⟩ '`<-`' ['`if`' ⟨*TypePremiseList*⟩
'`then`'] ⟨*TypeConsequent*⟩ '`return`' ⟨*InlineHaskell*⟩

⟨*TypeConsequent*⟩ ::= '`(`' ⟨*Identifier*⟩ ⟨*IdentifierList*⟩ '`)`' ['`:`' ⟨*Identifier*⟩
⟨*IdentifierList*⟩]
| '`[`' ⟨*Identifier*⟩ '`]`' [: ⟨*Identifier*⟩ ⟨*IdentifierList*⟩]
| '`[`' ⟨*Identifier*⟩ ⟨*Identifier*⟩ '`]`' [: ⟨*Identifier*⟩ ⟨*IdentifierList*⟩]
| '`[`' ⟨*Identifier*⟩ ⟨*Identifier*⟩ '`]`' [: ⟨*Identifier*⟩ ⟨*IdentifierList*⟩]

⟨*TypePremiseList*⟩ ::= ⟨*TypePremise*⟩
| ⟨*TypePremise*⟩ '`,`' ⟨*TypePremiseList*⟩

⟨*TypePremise*⟩ ::= ⟨*InlineHaskell*⟩
| '`(`' ⟨*Identifier*⟩ ⟨*Identifier*⟩ '`)`' ['`:`' ⟨*Identifier*⟩ ⟨*IdentifierList*⟩]
| '`[`' ⟨*Identifier*⟩ ⟨*Identifier*⟩ '`]`' ['`:`' ⟨*Identifier*⟩ ⟨*IdentifierList*⟩]

A.2 Inline Haskell Example

Listing A.1: Example of initial inline haskell code

```
1  {
2  import qualified Data.Map as M
3  import AbsAd
4
5  data ContextKey = Ret | Func Ident [Type] | Var Ident
6     deriving (Eq, Ord, Show, Read)
7
8  newtype Context = Context (Type, JetContextMap (Ident,
       [Type]) Type, JetContextMap Ident Type)
9     deriving (Eq, Ord, Show, Read)
10
```

```
11 instance JetContextBase Context where
12    emptyContext = Context (TNNone, emptyContext,
         emptyContext)
13    newBlock (Context (t, f, v) = Context (t, newBlock f,
         newBlock v)
14
15 instance JetContext Context ContextKey Type where
16    lookupContext Ret (Context (r, _, _, _, _)) = return r
17    lookupContext (Func ident ts) (Context (_, _, f, _, _)) =
         lookupContext (ident, ts) f
18    lookupContext (Var ident) (Context (_, _, _, v, _)) =
         lookupContext ident v
19    expandContext Ret t (Context (_, p, f, v, s)) = return
         (Context (t, p, f, v, s))
20    expandContext (Func ident ts) t (Context (rt, p, f, v,
         s)) = do
21       f' <- expandContextIf (\(i, t) (JetContextMap (b:bs))
            -> null (filter (\(i', t') -> i == i') (M.keys b)))
            (ident, ts) t f
22       return (Context (rt, p, f', v, s))
23    expandContext (Var ident) t (Context (rt, p, f, v, s)) =
         do
24       v' <- expandContext ident t v
25       return (Context (rt, p, f, v', s))
26
27 makeCheckError e t1 t2 = "Type␣error"
28 makeCheckErrorList _ _ _ = "Type␣error"
29 makeInferError _ _ = "Type␣error"
30 }
```

B Example Languages

B.1 Ad Language

B.1.1 AST

```
1  newtype Ident = Ident String deriving (Eq, Ord, Show, Read)
2  data Program = Prog [TypeDecl] ProcDecl
3
4  data TypeDecl = TDecl Ident TypeName type name = t
5
6  data FuncDecl = FDecl Ident [VarDecl] TypeName [Decl]
       [Stmnt] -- function name(params) return t decls being
       stmnts end
7
8  data ProcDecl = PDecl Ident [VarDecl] [Decl] [Stmnt] --
       procedure name(params) decls being stmnts end
9
10 data VarDecl = VDecl Ident TypeName -- name : t
11
12 data Decl = DFunc FuncDecl | DProc ProcDecl | DVar VarDecl
13
14 data Stmnt
15    = SNull -- null
16    | SPrint Expr -- print e
17    | SAssign [LValue] Expr -- name := e
18    | SIf Expr [Stmnt] [Stmnt] -- if e then stmnts1 else
          stmnts2 end
19    | SWhile Expr [Stmnt] -- while e do stmnts end
20    | SBlock [Decl] [Stmnt] -- declare decls begin stmnts end
21    | SCall Ident [Expr] -- name(params)
22    | SReturn Expr -- return e
23
24 data Expr
25    = EInt Integer
26    | ETrue -- false
27    | EFalse -- true
28    | EVar [LValue] -- name
29    | ECall Ident [Expr] -- name(params)
30    | ENeg Expr -- neg e
31    | ENot Expr -- not e
32    | EMul Expr Expr -- e1 * e2
33    | EAnd Expr Expr -- e1 and e2
34    | EAdd Expr Expr -- e1 + e2
35    | EOr Expr Expr -- e1 or e2
```

```
36      | EEq Expr Expr -- e1 = e2
37      | ELT Expr Expr -- e1 < e2
38
39  data LValue = LVVar Ident | LVArr LValue Expr -- name |
        name[e]
40
41  data TypeName
42     = TNBool -- bool
43     | TNInt -- int
44     | TNSyn Ident -- name
45     | TNArr TypeName -- array t
46     | TNRec [VarDecl] -- record decls end
47     | TNNone -- *internal*
```

B.1.2 Type Rules

The definition of Γ for the Ad language is a 5-tuple of the form (τ_{ret}, P, F, V, T), where τ_{ret} is the return type of the current subprogram, P is the name-space of procedures, F is the name-space for functions, V is the name space for variables, T is the name-space for types. The name-spaces P, F, V are block structured whereas T is not, therefore $newblock(\Gamma)$ is really $(\tau_{ret}, newblock(P), newblock(F), newblock(V), T)$

Statements

$$\text{AdTDeclT}\ \frac{expandContext(v, \tau, T) \vdash \lfloor decl \rfloor\ valid}{\Gamma \vdash \lfloor\ \text{type}\ v = \tau; decl \rfloor\ valid}$$

$$\text{AdVDecl}\ \frac{expandContext(v, \tau, V) \vdash \lfloor decl \rfloor\ valid}{\Gamma \vdash \lfloor\ \text{type}\ v : \tau; decl \rfloor\ valid}$$

$$\text{AdVDecl}\ \frac{\Gamma' \vdash \lfloor block \rfloor\ valid \qquad \Gamma' \vdash \lfloor decl \rfloor\ valid}{\Gamma \vdash \lfloor\ \text{procedure}\ v(p) block; decl \rfloor\ valid}\ [\Gamma' = expandContext((v, p), P)]$$

$$\text{AdVDecl}\ \frac{\Gamma', \tau_{ret} = \tau \vdash \lfloor block \rfloor\ valid \qquad \Gamma' \vdash \lfloor decl \rfloor\ valid}{\Gamma \vdash \lfloor\ \text{function}\ v(p)\ \text{return}\ \tau\ block; decl \rfloor\ valid}\ [\Gamma' = expandContext((v, p), F)]$$

Figure B.1: Type rules for declarations in the Ad language

$$\text{AdNullT}\ \frac{}{\Gamma \vdash \lfloor\ \mathsf{null}\ \rfloor\ valid} \qquad \text{AdPrintT}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lfloor\ \mathsf{print}\ e \rfloor\ valid}$$

$$\text{AdAssignT:}\ \frac{\Gamma \vdash e_l : t \qquad \Gamma \vdash e_r : t}{\Gamma \vdash \lfloor e_l := e_r \rfloor\ valid}\ [e_l \in \{e[e_{int}], e_{rec}.e, v\}]$$

$$\text{AdIfT:}\ \frac{\Gamma \vdash e : Bool \qquad \Gamma \vdash \lfloor s_1 \rfloor\ valid \qquad \Gamma \vdash \lfloor s_2 \rfloor\ valid}{\Gamma \vdash \lfloor\ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{end}\ \rfloor\ valid}$$

$$\text{AdWhileT:}\ \frac{\Gamma \vdash e : Bool \qquad \Gamma \vdash \lfloor s \rfloor\ valid}{\Gamma \vdash \lfloor\ \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{end}\ \rfloor\ valid}$$

$$\text{AdBlockT:}\ \frac{newBlock(\Gamma) \vdash \lfloor d \rfloor\ valid \qquad \Gamma' \vdash \lfloor s \rfloor\ valid}{\Gamma \vdash \lfloor\ \mathsf{declare}\ d\ \mathsf{begin}\ s\ \mathsf{end}\ \rfloor\ valid}\ \Gamma' = \Gamma \oplus d$$

$$\text{AdPCallT:}\ \frac{}{\Gamma \vdash \lfloor v(params) \rfloor\ valid}\ [(v, params) \in P)]$$

$$\text{AdRetT:}\ \frac{\Gamma \vdash e : \tau_{ret}}{\Gamma \vdash \lfloor\ \mathsf{return}\ e \rfloor\ valid}$$

Figure B.2: Type rules for statements in the Ad language

Expressions

$$\text{AdTrueT}\ \frac{}{\Gamma \vdash true : Bool} \quad \text{AdFalseT}\ \frac{}{\Gamma \vdash false : Bool}$$

$$\text{AdInt}\ \frac{}{\Gamma \vdash n : Int} \quad \text{AdNot}\ \frac{\Gamma \vdash e : Bool}{\Gamma \vdash not\ e : Bool} \quad \text{AdNeg}\ \frac{\Gamma \vdash e : Int}{\Gamma \vdash neg\ e : Int}$$

$$\text{AdAnd}\ \frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1\ and\ e_2 : Bool}$$

$$\text{AdMul}\ \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 * e_2 : Int}$$

$$\text{AdOr}\ \frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1\ or\ e_2 : Bool}$$

$$\text{AdAdd}\ \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 + e_2 : Int}$$

$$\text{AdEq}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : Bool} \quad \text{AdLT}\ \frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 < e_2 : Bool}$$

$$\text{AdFCallT:}\ \frac{}{\Gamma \vdash v(params) : \tau}\ [\tau = lookup((v, params), F)]$$

$$\text{AdArrAccT:}\ \frac{\Gamma \vdash e : Arr\ \tau \qquad \Gamma \vdash e_{int} : Int}{\Gamma \vdash e[e_{int}] : \tau}$$

$$\text{AdRecAccT:}\ \frac{\Gamma \vdash e : Rec\ D}{\Gamma \vdash e_1.v : \tau}\ [\tau = lookup(v, D)]$$

Figure B.3: Type rules for expressions in the Ad language

Type Synonyms

$$\text{AdSynT:}\ \frac{\Gamma \vdash \tau : Syn\ v}{\Gamma \vdash \tau : \sigma}\ [\sigma = lookup(v, T)] \quad \text{AdT:}\ \frac{}{\Gamma \vdash \tau : \tau}$$

Figure B.4: Type rule for type synonyms

B.1.3 Ad JET Specification

```
1  {
2  import qualified Data.Map as M
3  import AbsAd
4
5  type Type = TypeName
6
7  data ContextKey = Ret | Proc Ident [Type] | Func Ident
       [Type] | Var Ident | Typ Ident
8     deriving (Eq, Ord, Show, Read)
```

```
 9
10 newtype Context = Context (Type, JetContextMap (Ident,
       [Type]) Type, JetContextMap (Ident, [Type]) Type,
       JetContextMap Ident Type, JetContextMap Ident Type)
11     deriving (Eq, Ord, Show, Read)
12
13 instance JetContextBase Context where
14     emptyContext = Context (TNNone, emptyContext,
           emptyContext, emptyContext, emptyContext)
15     newBlock (Context (t, p, f, v, s)) = Context (t, newBlock
           p, newBlock f, newBlock v, s)
16
17 instance JetContext Context ContextKey Type where
18     lookupContext Ret (Context (t, _, _, _, _)) = Succ t
19     lookupContext (Proc ident ts) (Context (_, p, _, _, _)) =
           lookupContext (ident, ts) p
20     lookupContext (Func ident ts) (Context (_, _, f, _, _)) =
           lookupContext (ident, ts) f
21     lookupContext (Var ident) (Context (_, _, _, v, _)) =
           lookupContext ident v
22     lookupContext (Typ ident) (Context (_, _, _, _, s)) =
           lookupContext ident s
23     expandContext Ret t' (Context (_, p, f, v, s)) = Succ
           (Context (t', p, f, v, s))
24     expandContext (Proc ident ts) t (Context (rt, p, f, v,
           s)) = do
25         p' <- expandContextIf (\(i, t) (JetContextMap (b:bs))
               -> null (filter (\(i', t') -> i == i') (M.keys b)))
               (ident, ts) t p
26         Succ (Context (rt, p', f, v, s))
27     expandContext (Func ident ts) t (Context (rt, p, f, v,
           s)) = do
28         f' <- expandContextIf (\(i, t) (JetContextMap (b:bs))
               -> null (filter (\(i', t') -> i == i') (M.keys b)))
               (ident, ts) t f
29         Succ (Context (rt, p, f', v, s))
30     expandContext (Var ident) t (Context (rt, p, f, v, s)) =
           do
31         v' <- expandContext ident t v
32         Succ (Context (rt, p, f, v', s))
33     expandContext (Typ ident) t (Context (rt, p, f, v, s)) =
           do
34         s' <- expandContext ident t s
35         Succ (Context (rt, p, f, v, s'))
36
37 makeCheckError e t1 t2 = "Type␣error"
38 makeCheckErrorList _ _ _ = "Type␣error"
39 makeInferError _ _ = "Type␣error"
40
41 vdeclsTypes :: Functor f => f VarDecl -> f Type
42 vdeclsTypes = fmap (\(VDecl _ t) -> t)
43 }
44
45 typerule ProgT <- if {ctx} |- [TypeDecl tdecls], {var1} |-
       (ProcDecl proc) then (Program Prog tdecls proc) return
```

```
      {Succ ()};
46
47  typerule TypeDeclT <- if {ctx} |- (Type t) : t', {ctx' <-
       expandContext (Typ name) t' ctx} then (TypeDecl TDecl
       name t) return {Succ ctx'};
48
49  typerule TypeDeclEmpty <- [TypeDecl] return {Succ ctx};
50  typerule TypeDeclCons <- if {ctx} |- (TypeDecl decl), {var1}
       |- [TypeDecl decls] then [TypeDecl decl decls] return
       {Succ var2};
51
52  typerule FuncDeclT <- if
53     {ctx' <- expandContext (Func name (vdeclsTypes vdecls)) t
          ctx},
54     {(newBlock ctx')} |- [VarDecl vdecls],
55     {var1} |- [Decl decls],
56     {ctxret <- expandContext Ret t var2},
57     {ctxret} |- [Stmnt stmnts] then (FuncDecl FDecl name
          vdecls t decls stmnts) return {Succ ctx'};
58  typerule ProcDeclT <- if
59     {ctx' <- expandContext (Proc name (vdeclsTypes vdecls))
          TNNone ctx},
60     {(newBlock ctx')} |- [VarDecl vdecls],
61     {var1} |- [Decl decls],
62     {ctxret <- expandContext Ret TNNone var2},
63     {ctxret} |- [Stmnt stmnts] then (ProcDecl PDecl name
          vdecls decls stmnts) return {Succ ctx'};
64  typerule VarDeclT <- if {ctx} |- (Type t) : t', {ctx' <-
       expandContext (Var name) t' ctx} then (VarDecl VDecl name
       t) return {Succ ctx'};
65
66  typerule VarDeclEmpty <- [VarDecl] return {Succ ctx};
67  typerule VarDeclCons <- if {ctx} |- (VarDecl decl), {var1}
       |- [VarDecl decls] then [VarDecl decl decls] return {Succ
       var2};
68
69  typerule DFuncT <- if {ctx} |- (FuncDecl decl) then (Decl
       DFunc decl) return {Succ var1};
70  typerule DProcT <- if {ctx} |- (ProcDecl decl) then (Decl
       DProc decl) return {Succ var1};
71  typerule DVar <- if {ctx} |- (VarDecl decl) then (Decl DVar
       decl) return {Succ var1};
72
73  typerule DeclEmpty <- [Decl] return {Succ ctx};
74  typerule DeclCons <- if {ctx} |- (Decl decl), {var1} |-
       [Decl decls] then [Decl decl decls] return {Succ var2};
75
76  typerule NullT <- (Stmnt SNull) return {Succ ()};
77  typerule PrintT <- if {ctx} |- (Expr e) : t then (Stmnt
       SPrint e) return {Succ ()};
78  typerule AssignT <- if {ctx} |- [LValue name] : t, {ctx} |-
       (Expr e) : t then (Stmnt SAssign name e) return {Succ ()};
79  typerule IfT <- if {ctx} |- (Expr e) : TNBool, {ctx} |-
       [Stmnt s1], {ctx} |- [Stmnt s2] then (Stmnt SIf e s1 s2)
       return {Succ ()};
```

```
80  typerule WhileT <- if {ctx} |- (Expr e) : TNBool, {ctx} |-
        [Stmnt stmnts] then (Stmnt SWhile e stmnts) return {Succ
        ()};
81  typerule BlockT <- if {(newBlock ctx)} |- [Decl decls],
        {var1} |- [Stmnt stmnts] then (Stmnt SBlock decls stmnts)
        return {Succ ()};
82  typerule PCallT <- if {ctx} |- [Expr params] : ts,
        {lookupContext (Proc name ts) ctx :: JetError Type} then
        (Stmnt SCall name params) return {Succ ()};
83  typerule RetT <- if {ctx} |- (Expr e) : t, {tret <-
        lookupContext Ret ctx}, {if tret == t then Succ () else
        Fail "Type␣error"} then (Stmnt SReturn e) return {Succ
        ()};
84
85  typerule StmntEmpty <- [Stmnt] return {Succ ()};
86  typerule StmntCons <- if {ctx} |- (Stmnt stmnt), {ctx} |-
        [Stmnt stmnts] then [Stmnt stmnt stmnts] return {Succ ()};
87
88  typerule IntT <- (Expr EInt n) : TNInt return {Succ ()};
89  typerule TrueT <- (Expr ETrue) : TNBool return {Succ ()};
90  typerule FalseT <- (Expr EFalse) : TNBool return {Succ ()};
91  typerule VarT <- if {ctx} |- [LValue name] : t then (Expr
        EVar name) : t return {Succ ()};
92  typerule NegT <- if {ctx} |- (Expr e) : TNInt then (Expr
        ENeg e) : TNInt return {Succ ()};
93  typerule NotT <- if {ctx} |- (Expr e) : TNBool then (Expr
        ENot e) : TNBool return {Succ ()};
94  typerule ECallT <- if {ctx} |- [Expr params] : ts, {t <-
        lookupContext (Func name ts) ctx} then (Expr ECall name
        params) : t return {Succ ()};
95  typerule MulT <- if {ctx} |- (Expr e1) : TNInt, {ctx} |-
        (Expr e2) : TNInt then (Expr EMul e1 e2) : TNInt return
        {Succ ()};
96  typerule AndT <- if {ctx} |- (Expr e1) : TNBool, {ctx} |-
        (Expr e2) : TNBool then (Expr EAnd e1 e2) : TNBool return
        {Succ ()};
97  typerule AddT <- if {ctx} |- (Expr e1) : TNInt, {ctx} |-
        (Expr e2) : TNInt then (Expr EAdd e1 e2) : TNInt return
        {Succ ()};
98  typerule OrT <- if {ctx} |- (Expr e1) : TNBool, {ctx} |-
        (Expr e2) : TNBool then (Expr EOr e1 e2) : TNBool return
        {Succ ()};
99  typerule EqT <- if {ctx} |- (Expr e1) : t, {ctx} |- (Expr
        e2) : t then (Expr EEq e1 e2) : TNBool return {Succ ()};
100 typerule LTT <- if {ctx} |- (Expr e1) : TNInt, {ctx} |-
        (Expr e2) : TNInt then (Expr ELT e1 e2) : TNBool return
        {Succ ()};
101 typerule LVVarT <- if {t <- lookupContext (Var name) ctx}
        then (LValue LVVar name) : t return {Succ ()};
102 typerule LVArrT <- if {ctx} |- (LValue name) : TNArr t,
        {ctx} |- (Expr e) : TNInt then (LValue LVArr name e) : t
        return {Succ ()};
103
104 typerule LValueSngle <- if {ctx} |- (LValue name) : t then
        [LValue name] : t return {Succ ()};
```

```
105 typerule LValueCons <- if {let Context (tret, _, _, _,
       tdecls) = ctx; ctx' = Context (tret, emptyContext,
       emptyContext, emptyContext, tdecls)},
106       {ctx} |- (LValue nhead) : tmp, {ctx} |- (Type tmp) :
              TNRec decls, {ctx'} |- [VarDecl decls], {var1} |-
              [LValue ntail] : t then
107    [LValue nhead ntail] : t return {Succ ()};
108
109 typerule ExprEmpty <- if {let ts = []} then [Expr] : ts
       return {Succ ()};
110 typerule ExprCons <- if {ctx} |- (Expr e) : t, {ctx} |-
       [Expr es] : ts, {let ts' = t : ts} then [Expr e es] : ts'
       return {Succ ()};
111
112 typerule SynT <- if {t' <- lookupContext (Typ t) ctx} then
       (Type TNSyn t) : t' return {Succ ()};
113 typerule TypeT <- (Type t) : t return {Succ ()};
```

B.2 Simply Typed Lambda Calculus

Γ in the Simply Typed Lambda Calculus is simply just a single name-space containing variables.

B.2.1 AST

```
 1 data Id = Id String
 2
 3 data Literal
 4    = LitInt Integer
 5    | LitBool Bool
 6
 7 data Expr
 8    = Var Id
 9    | Lit Literal
10    | App Expr Expr -- e1 e2
11    | Lam Id Type Expr -- \ name : t . e
12    | If Expr Expr Expr -- \ if e1 then e2 else e3
13    | Fix Expr -- fix e
14    | Pred Expr -- pred e
15    | ESucc Expr -- succ e
16    | Iszero Expr -- iszero e
17
18 data Type
19    = TInt - Int
20    | TBool - Bool
21    | TFun Type Type - t1 -> t2
```

B.2.2 Type Rules

$$\text{STLCBool: } \frac{}{\Gamma \vdash b : Bool}\,[b \in \{true, false\}]$$

$$\text{STLCInt: } \frac{}{\Gamma \vdash n : Int}\,[n \in \mathbb{N}]$$

$$\text{STLCVar: } \frac{}{\Gamma \vdash n : \tau}\,[\tau = lookup(x, \Gamma)]]$$

$$\text{STLCLam: } \frac{expandContext(x, \sigma, newblock(\Gamma)) \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma \,.\, e) : (\sigma \to \tau)}$$

$$\text{STLCApp: } \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\text{STLCPred: } \frac{\Gamma \vdash e : Int}{\Gamma \vdash pred\ e : Int} \quad \text{STLCSucc: } \frac{\Gamma \vdash e : Int}{\Gamma \vdash succ\ e : Int}$$

$$\text{STLCIszero: } \frac{\Gamma \vdash e : Int}{\Gamma \vdash iszero\ e : Bool} \quad \text{STLCFix: } \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash fix\ e : \tau}$$

$$\text{STLCIf: } \frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau}$$

Figure B.5: Type rules for the simply typed lambda calculus

B.2.3 JET Specification

```
1  {
2  import Lang.Ast
3
4  type Context = JetContextMap Id Type
5  makeCheckError _ _ _ = "Type␣Error"
6  makeInferError _ _ = "Type␣Error"
7  }
8
9  typerule TLitBool <- (Literal LitBool b) : TBool return
       {return ()};
10 typerule TLitInt <- (Literal LitInt n) : TInt return {return
       ()};
11
12 typerule TExprVar <- if {t <- lookupContext var ctx} then
       (Expr Var loc var) : t return {return ()};
13 typerule TExprLit <- if {ctx} |- (Literal k) : t then (Expr
       Lit pos k) : t return {return ()} ;
14 typerule TExprIf <- if {ctx} |- (Expr ep) : TBool, {ctx} |-
       (Expr et) : t, {ctx} |- (Expr ef) : t then (Expr If ep et
       ef) : t return {return ()};
```

```
15 typerule TExprApp <- if {ctx} |- (Expr e1) : TFun t1 t2,
       {ctx} |- (Expr e2) : t1 then (Expr App e1 e2) : t2 return
       {return ()};
16 typerule TExprLam <- if {=<< expandContext var t1 (newBlock
       ctx)} |- (Expr e) : t2 then (Expr Lam var t1 e) : TFun t1
       t2 return {return ()};
17 typerule TExprFix <- if {ctx} |- (Expr e) : TFun t1 t2, {if
       t1 == t2 then return () else fail "Type␣error"} then
       (Expr Fix e) : t1 return {return ()};
18 typerule TExprPred <- if {ctx} |- (Expr e) : TInt then (Expr
       Pred e) : TInt return {return ()};
19 typerule TExprSucc <- if {ctx} |- (Expr e) : TInt then (Expr
       ESucc e) : TInt return {return ()};
20 typerule TExprIszero <- if {ctx} |- (Expr e) : TInt then
       (Expr Iszero e) : TBool return {return ()};
```

C Haskell Modules

C.1 Jet Context Base

```
1  class JetContextBase t where
2     emptyContext :: t
3     newBlock :: t -> t
```

C.2 Jet Context Relational

```
1  class JetContextBase r => JetContext r k t where
2     lookupContext :: Ord k => k -> r -> JetError t
3     expandContext :: Ord k => k -> t -> r -> JetError r
4     expandContextIf :: Ord k => (k -> r -> Bool) -> k -> t ->
          r -> JetError r
```

C.2.1 Jet Context Map

```
1  instance JetContextBase (JetContextMap a b) where
2     emptyContext = JetContextMap [M.empty]
3     newBlock (JetContextMap blocks) = JetContextMap (M.empty
          : blocks)
4
5  instance JetContext (JetContextMap a b) a b where
6     lookupContext key (JetContextMap blocks) = maybe2Error
          "Lookup␣error" $ M.lookup key (M.unions blocks)
7     expandContext key t (JetContextMap (block:blocks)) = do
8        block' <- insertUnique key t block
9        return (JetContextMap (block':blocks))
10    expandContextIf predicate key t ctx =
11       if predicate key ctx then expandContext key t ctx
12       else fail "Predicate␣check␣failed"
```

C.3 Jet Error Monad

```
1  module JetErrorM where
2
3  import Control.Monad
4  import Control.Monad.Fail
5
```

```
 6 data JetError a = Succ a | Fail String deriving (Show, Read,
       Eq, Ord)
 7
 8 instance Functor JetError where
 9    fmap = liftM
10
11 instance Applicative JetError where
12    pure = Succ
13    (<*>) = ap
14
15 instance Monad JetError where
16    return = pure
17    fail = Fail
18    Succ a >>= f = f a
19    Fail s >>= _ = Fail s
20
21 instance MonadFail JetError where
22    fail = Fail
23
24 maybe2Error :: String -> Maybe a -> JetError a
25 maybe2Error s Nothing = Fail s
26 maybe2Error _ (Just a) = Succ a
```