An ML Implementation of the Dependently Typed Lambda Calculus

Samuel Baxter
2014 Honors Thesis
Advised by Professor Robert Muller
Computer Science Department, Boston College

May 14, 2014

$\lambda\Pi$

Abstract: As programming languages and the field of computer science develop, the question of program correctness and reliability becomes more prevalent in our field. How can we assure that programs execute as expected, and what errors can we catch before we reach them? Type systems provide a framework to give satisfactory answers to these questions, and more expressive type systems yield more reliable conclusions drawn from static analysis of a program. However, more complex systems are computationally expensive, so the difficulty lies in striking a balance between a type system's effectiveness and its practicality in implementation and in execution.

This thesis discusses the limits of simple type systems as well as the polymorphic lambda calculus that underlies Java's Generics. We provide an Ocaml implementation of a dependently typed, higher-order functional language, to display the benefits of this expressive system that catches errors before runtime many other systems cannot detect. This aims to be a simple, general ML implementation that does not rely on any language-specific features, so as to highlight the type system in its clearest form. The dependently typed lambda calculus is compared to other systems, with concrete examples and a discussion of its benefits and drawbacks.

Contents

1 Introduction

Errors are the unavoidable hurdles of the computer programming experience. From forgetting a semi-colon on a first 'Hello World' program, to runtime segmentation faults caused by improper user input, computer scientists will always face the task of debugging their programs, and will never escape the suspicion that a user can make the whole program crash with one illegal command. Computer scientists are faced with the task of writing programs that perform expected computations, but what means do we have to ensure that programs actually execute accordingly to these expectations? Many people disagree on the best methods for ensuring proper program execution, whether they support unit testing, static analysis, or another technique, and programming languages themselves are designed with different strategies for handling these issues.

Type systems provide a formal framework for automating a portion of program analysis, defining typing rules for a type checking algorithm that processes a program to detect type errors. These systems associate a formal semantics with the abstract syntax of a programming language, specifying how different language constructs are allowed to interact with each other. Because this approach analyzes an intermediate representation of the program, usually in the form of an abstract syntax tree generated by parsing the program text, type checking can be performed before program execution, and even before compilation. Some languages apply this type checking algorithm and only compile programs that satisfy the type system without producing a type error, catching many errors before execution that would otherwise cause the program to terminate unexpectedly.

This thesis explores type systems in depth, with the goal of highlighting the insufficiencies of many widely used languages, and compares the costs and benefits gained from more complex type systems. Various type theories are presented and discussed, with the goal of providing enough theoretical background to make the formal theory of dependent lambda calculus accessible.

1.1 Introduction to Type Systems

1.1.1 Statically Typed Languages

Programming languages are generally classified into one of two categories, statically or dynamically typed. Languages that are statically typed, such as Java, C/C++, and ML-like languages, apply a type checking operation during the compilation process, and only compile a program if type

checking does not detect any type errors in the code. Errors such as multiplying an integer by a string, or adding an integer to an array defined to be an array of some user defined object, are caught by the type checker before compilation, whereas other languages may not detect these errors until program execution. This adds a level of safety to the programming language, as the user gains confidence that a program that passes the type checker will execute as it is expected.

However, static type checking comes with some drawbacks as well. As this process must be performed before compiling, compilation becomes more expensive. This does allow for some runtime optimizations that yield more efficient code, but it is also costly to compile and more difficult to write well-typed programs than in dynamic languages that allow much more flexibility. The added restrictions in a statically typed language occasionally rejects some programs that would execute properly occasionally fail to type check. In the case of if-statements in a statically typed language such as Java, the type system necessitates that the *then* and *else* clauses have the same type, but a program such as

```
if (True)
   return 1;
else
   return "This case will never happen";
```

fails to type check even though the *else* clause is never reached in execution. Many programmers begrudge these systems for this reason, but proponents of static typing think the limitations are worth the reward.

1.1.2 Dynamically Typed Languages

Often referred to as 'untyped' languages, some languages such as Python do not type check until a program is actually executed. These dynamic systems therefore do not catch type errors until runtime, as there is no static analysis of the program. Because runtime errors are typically fatal, something as easily caught as applying the '+' operator to anything other than a number can go unnoticed until suddenly causing the whole program to fail. But dynamically typed languages allow programmers much more flexibility and power in writing code, as when used properly, complex features such as polymorphism are trivially handled by the runtime system of the language.

1.2 The Simply Typed Lambda Calculus

Beyond determining when a language should handle type checking, when writing a programming language, we must also decide what forms of typing we want to allow. In a static environment, types are checked at compile time, but do programs require explicit type annotations? Should functions be passable values? There exist different theories on which to base these language designs, and this decision carries significant influence over what a programming lanuage allows, or considers an error.

1.2.1 Syntax

The simply typed lambda calculus, $\lambda_{\rightarrow}$, is the theoretical basis for typed, functional programming languages [3]. This is typically the first formal system one learns about when studying type theory, and most typed languages handle type checking similarly to $\lambda_{\rightarrow}$, so it makes sense for us to first look at this calculus before considering more complex systems. $\lambda_{\rightarrow}$ is the smallest statically typed, functional programming language we can imagine. Terms in the language can only be one of four forms: variables, constants, abstractions (anonymous functions), or applications. Types are even more simple, and can only be either a base type α or a function between types.

e	$:=$	x	Variable
	$\mid$	c	Constant
	$\mid$	$\lambda x{:}\tau.e$	Lambda Abstraction
	$\mid$	$e\ e$	Application
τ	$:=$	α	Base Type
	$\mid$	$\tau \rightarrow \tau$	Function Type

Figure 1: Abstract Syntax of $\lambda_{\rightarrow}$

As an example, we can derive the expression

$$(\lambda x{:}int.x)\ 7$$

by following the steps

$$e \mapsto e\ e \mapsto (\lambda x{:}\tau.e)\ e \mapsto (\lambda x{:}int.e)\ e \mapsto (\lambda x{:}int.x)\ e \mapsto (\lambda x{:}int.x)\ 7$$

where x is a variable, int is a base type, and 7 is a constant in this calculus.

1.2.2 Type Checking

$\lambda_{\rightarrow}$ associates type semantics to the syntax described in the previous section. It is important to note that there is also an evaluation semantics associated with these caculi, but in the sake of emphasizing the type systems themselves, evaluation will be omitted. The following typing rules, with respect to a context Γ containing type information of variables and constants, assert that if a program is typed properly, then the program should be evaluated without any unexpected errors [3].

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : \tau \rightarrow \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}\ (\textsc{App})$$

$$\frac{\Gamma[x := \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : \tau \rightarrow \tau'}\ (\textsc{Lam}) \qquad \frac{\Gamma(c) = \tau}{\Gamma \vdash c : \tau}\ (\textsc{Const})$$

Figure 2: Type Rules for $\lambda_{\rightarrow}$

Type rules of the form $\Gamma \vdash e{:}\tau$ indicate that a term e, derivable in the syntax of $\lambda_{\rightarrow}$, has type τ in context Γ, where τ is a type derivable by the syntax of types in $\lambda_{\rightarrow}$. Note that according to these rules, $\lambda_{\rightarrow}$ is not polymorphic, that is, a type identifier τ represents one specific type, and cannot be abstracted over.

When we type check a variable, according to the (Var) rule, we first look up the variable x in the context Γ. If $x \in \Gamma$ is mapped to τ, then we say that x has type τ. The typing rule for constants works in a similar fashion to variables. For lambda abstractions (Lam), we first extend the context Γ to associate the variable x with type τ. If in this new context, e has type τ', we say that the lambda expression has type $\tau \rightarrow \tau'$. For application (App), we first address the function being applied. If e has the function type $\tau \rightarrow \tau'$, and e' has type τ, then the result of applying e to e' is the codomain of the type of e, that is, τ'. If the type rule is not satisfied in any of these cases, the expression fails to type check and we do not proceed with compilation or program execution.

1.2.3 Example Type Derivation

To clarify how we apply these type rules to an expression, let us observe the earlier example:

$$(\lambda x{:}int.x)\ 7$$

Assuming we are within a context Γ where *int* is a base type, and integers are constants in the language with type *int*, we can derive the type of this expression as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma[x \coloneqq int](x) = int}{\Gamma[x \coloneqq int] \vdash x : int}\ (\textsc{Var})}{\Gamma \vdash (\lambda x{:}int.x) : int \to int}\ (\textsc{Lam}) \qquad \dfrac{\Gamma(7) = int}{\Gamma \vdash 7 : int}\ (\textsc{Const})}{\Gamma \vdash (\lambda x{:}int.x)\ 7 : int}\ (\textsc{App})$$

In order to derive the type of the resulting expression, we first check that the applicant has a valid function type, via the (App) rule. We extend the context Γ to contain the domain information, that is that x has type *int*, and check the type of the body of the lambda expression by looking up the variable x in this new context. We confirm that the type of the applicant is $int \to int$, and then we proceed to look up the type of the operand, 7, according to the (Const) rule. As 7 has the desired type *int*, the resulting type of the application is type *int*.

1.3 System F

As was mentioned in the previous section, $\lambda_{\to}$ restricts type expressions τ to only represent a specific type, meaning polymorphic code is unachievable in a language based on that formal system. In order to write reusable code that works properly over all types, we need to extend the definition of the lambda calculus to allow abstraction over types.

System F provides such a solution, allowing lambda abstractions to range over types, as well as over values [3]. For simplicity's sake, we will only discuss relevant additional features to this system instead of going into the same level of detail as with $\lambda_{\to}$.

While all valid $\lambda_{\to}$ expressions and types are valid in System F, we also extend the abstract syntax to allow types as terms, and terms of the form

$$\Lambda\tau.e$$

and extend the syntax of types to allow types of the form

$$\forall\tau.\tau$$

This extension is the basis of the form of parametric polymorphism in Java Generics and the type systems of ML-like languages such as Standard ML (SML) and Ocaml. To highlight the strenght and flexibility of this formal system, we can compare the polymorphic identity function in System F to

the identity function in $\lambda_{\rightarrow}$. If we have a language with *int* and *String* as base types based off of the type rules of $\lambda_{\rightarrow}$, we must write separate identity functions $\lambda x{:}int.x$ and $\lambda x{:}String.x$, with types $int{\rightarrow}int$ and $String{\rightarrow}String$ respectively, to operate on expressions of the two different base types. Although these are perfectly valid in $\lambda_{\rightarrow}$, this creates redundant code that could be significantly condensed with a more powerful system, as the two functions perform equivalent functions.

In System F, we write the polymorphic identity function as

$$\Lambda\alpha.\lambda x{:}\alpha.x$$

This allows us to abstract over the type of the input, and we supply the desired type as the first parameter when we apply the function. That is, if we desire the identity function on objects of type *int*, we simply apply $\Lambda\alpha.\lambda x{:}\alpha.x$ to *int*, and replace all α's in the body of the Λ with *int*, yielding $\lambda x{:}int.x$. Thus the single polymorphic lambda abstraction can capture the identity function on any desired type, by simply providing the desired type as a parameter. We must adapt the type rules to allow for application of abstractions to types by the following:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ (\text{T-Lam}) \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau' \qquad \Gamma \vdash \tau : \text{Type}}{\Gamma \vdash e\ \tau : \tau'[\alpha \coloneqq \tau]}\ (\text{T-App})$$

Figure 3: Extended Type Rules for System F

where $\tau'[\alpha \coloneqq \tau]$ in the (T-App) rule means that we replace all occurrences of α in τ' with τ, and the assertion $\Gamma \vdash \tau : \text{Type}$ means that τ is a valid type in context Γ. Thus we can provide a type judgement for the polymorphic identity function as follows:

$$\cfrac{\cfrac{\cfrac{\Gamma[x \coloneqq \alpha](x) = \alpha}{\Gamma[x \coloneqq \alpha] \vdash x : \alpha}\ (\text{Var})}{\Gamma \vdash (\lambda x{:}\alpha.x) : \alpha \rightarrow \alpha}\ (\text{Lam})}{\Gamma \vdash (\Lambda\alpha.\lambda x{:}\alpha.x) : \forall\alpha.\alpha \rightarrow \alpha}\ (\text{T-Lam})$$

and conclude that the polymorphic identity function in System F has type $\forall\alpha.\alpha \rightarrow \alpha$.

As we see in this example, System F allows programmers much more flexibility and power to write reusable, polymorphic code. This lies at the basis of polymorphic data structures, map, and fold functions that make code efficient and take advantage of properties of functional languages to their fullest.

1.4 Improvements

In languages implementing the type theories describe up to this point, type checking can catch errors at runtime such as applying a function to an input of the wrong type, or trying to append an element of the wrong type to a typed list. Static type checking provides a huge benefit to programmers by catching many mistakes before a program is ran, preventing runtime crashes. But even in languages based off of the polymorphic System F, there are runtime errors we wish we could prevent through static analysis. While it is arguable that the compile-time costs and restricted freedom on the side of the programmer make type systems more of a hindrance than beneficial, we can extend these type systems further to catch even more errors before runtime, increasing the value of static analysis.

Accompanying this thesis is an OCaml implementation of a dependently typed, higher-order functional programming language, based off of the dependently typed lambda calculus, $\lambda\Pi$, developed from Per Martin Löf's Intuitionistic Type Theory in 1972. Informally, this system proposes to be able to catch many errors missed by other type theories at compile time by incorporating highly expressive types that contain more information than simple type systems. Similar to System F, $\lambda\Pi$ allows for polymorphsim, but in a much more powerful manner, by allowing interaction between types and terms, rather than them having separate abstractions. $\lambda\Pi$ introduces a hierarchy of types, or type universes, where we say that a type τ has kind $\star$, and a term e still has type τ. With this addition, $\lambda\Pi$ extends System F to having not only polymorphic types and type abstractions, but allowing types to depend on values of expressions, as well as on the types of those expressions.

2 Dynamically Typed Languages

To provide a comparison of how different popular languages handle typing situations, we progress with examples of Python code, and a discussion of the flexibility it allows, and the potential drawbacks of this dynamically typed programming environment.

2.1 Python

Python is a dynamically typed language, that supports many functional programming features, despite not adhering to a static type system. Types and values have the same status in the language, similar to $\lambda\Pi$, and func-

tions are first-class citizens, meaning they can be passed as arguments, and returned as output of other functions. Although some of these features may appear similar to the formal systems discussed, we will see how Python differs in handling program semantics and execution.

2.1.1 Code Example

As briefly presented earlier, Python allows the *then* and *else* clauses of if-statements to differ in their types. While programmers contend that statically typed languages may reject programs that would never fail to execute, the lack of restrictions in a language like Python could be the root of easily preventable, fatal errors at runtime.

To highlight the behavior of if-statements in a dynamically typed program, let's observe the following factorial function:

```
def factorial(n):
    if n < 0:
        return "Invalid Input"
    elif n == 0:
        return 1
    else :
        return n * factorial(n-1)
```

If `factorial` is called with a non-negative input, the function performs the expected factorial computation we are familiar with from mathematics. However, Python allows us to handle negative inputs in any way we want, whereas statically typed languages could only produce an exception to be handled or a runtime error. Here, the function returns a value of type *String* when called with a negative number. This does allow programmers to return meaningful information while avoiding a fatal error, but to advocates of type systems, no longer knowing how the program will execute is too dangerous a risk to take.

Additionally, nothing necessitates calling `factorial` with an integer, so we can just as easily call it with a floating-point number. Again, we do not know how the program will behave on an atypical input, as there is no standard mathematical definition of the factorial function on non-integers. In a statically typed system, type annotations can require integer inputs, and promise integer outputs, and otherwise catch these errors before execution and fail to compile.

2.1.2 Polymorphism

As was just mentioned, Python would allow us to call `factorial` on integers, floating-point numbers, or an object of any other type, without complaining until possibly producing a runtime error. For the `factorial` example, this does not make the most sense, as the factorial function is only defined in mathematics on integer values. But this does display the flexibility and freedom a programmer has working in a dynamically typed environment. Though the factorial function may not take advantage of this freedom, untyped languages make writing polymorphic code nearly effortless.

As our first example, Python allows us to write the identity function as follows:

```
def identity(x):
    return x
```

which is polymorphic over all types of inputs. As the function's operation does not depend on any type-specific features of the input, it should naturally be polymorphic, but a language with explicit, static type declarations would require us to write nearly identical `identity` functions to operate on different types of input.

```
def n_copies(n, x):
    return [x]*n

def length(l):
    if l == []:
        return 0
    else :
        return 1 + length(l[1:])

def reverse(l):
    if length(l) == 0:
        return []
    return [l[-1]] + reverse(l[:-1])
```

Figure 4: Polymorphic Functions on Lists in Python

For a more interesting example, let us look at polymorphic operations on lists in Figure 4. Some features of lists are independent of the literal elements, or types of those elements, contained within a list. Creating a list of 'n' copies of an object, calculating the length of a list, or reversing the order of a list, are all naturally expressed as polymorphic functions.

Although these functions are polymorphic in the sense that they will properly execute on lists of any type of element, these are not type-safe according to any formal polymorphic type system. While we can reverse a list of integers, or a list of strings just as easily with this `reverse` function, there is nothing stopping an unknowing programmer from calling the reverse function on for example an integer or dictionary, rather than a list. The dynamic typing of Python enables these polymorphic functions to be written easily, but does nothing to ensure that they are used properly, as such a type error wouldn't be detected until runtime, crashing whatever program makes the illegal function call.

2.2 Runtime Errors

We have been discussing the shortcomings of dynamically typed programming languages for the past few sections, highlighting how errors preventable by static analysis become runtime errors that are often fatal to the program's execution. However, it is important to point out that some errors are by their nature runtime errors in most common type systems. While array or list indices, or invalid user input at runtime cannot be handled by languages like Java either, the issue with dynamic languages is that all errors become runtime errors. This puts additional pressure on programmers to write proper code, without providing any assistance to understand what the code actually does.

3 Statically Typed Languages

Now we switch our focus to statically typed languages with specific examples of Java programs. Programming takes on additional effort, however minimal, as explicit type annotations are required to satisfy the type checking algorithm, but we will see how Java's type system ensures that we are programming properly to the best of its ability.

3.1 Java

Java's type system at a basic level works in the same way as the simply typed lambda calculus. Programmers supply explicit type annotations to variables during assignment, and if the type of the expression is of anything other than that type, Java complains and halts compilation. Using an operator on an object of the wrong type, calling a function on mis-typed inputs, or defining a function to return a type contrary to its explicitly declared return type are all catchable errors during compilation, hence the statically typed nature of Java.

3.2 Java Generics

The fundamentals of a typed language should be commonly understood, so we will look at polymorphism in Java to highlight its differences from a dynamic environment like previously discussed.

Java Generics are based off of the polymorphic lambda calculus, System F, introduced earlier. This form of parametric polymorphism in Java has been supported since 2004, after the incorporation of Philip Wadler's Generic Java into J2SE 5.0. To briefly present some generic Java code, we can define a polymorphic List class as follows:

```
public class ListC<Item> implements List<Item> {
   private Node first;
   private class Node {
       Item info;
       Node next;
       public Node(Item info, Node  next) {
           this.info = info;
           this.info = info;
       }
   }
   public void add(Item i) {...}
   public Item get(int i) {...}
   public int find(Item i) {...}
   ⋮
}
```

Figure 5: Generic List Class Implemented in Java

This straightforward implementation of a List class is a syntactically nicer version of System F, where `Item` takes the place of all α in the calculus. When we instantiate a ListC of integers by declaring

```
List<Integer> list = new ListC<Integer>();
```

we replace all occurences of `Item` in the class with the type `Integer`. Now the `add` function accepts an `Integer` as input instead of an `Item`, so if we try to compile the line

```
list.add("This is a String, not an Integer...");
```

we will get a type error during compilation. Python would allow such actions, as lists are general and do not restrict the types of elements allowed in a list, that is, a list could contain both an integer and a string in Python. Here, by parameterizing `ListC` with the type `Integer`, we restrict the allowed types of elements to only integers.

To portray the polymorphism of Generics, we could just as easily instantiate a list strings by declaring

```
List<String> list_2 = new ListC<Integer>();
```

in which case we could add the string from before without producing a type error at compile time.

Java Generics allow programmers to write polymorphic, reusable code that ranges over all types of objects, and adds an additional level of type-safety for such programs that languages like Python cannot capture. This polymorphism is based off of the formal System F, and adheres to its theoretical type rules.

3.3 Deficiencies

Even with a more robust type system (or the mere existence of a static type system as compared to dynamically typed languages), languages such as Java cannot capture all errors during compilation. Notably, Java has no way of statically detecting illegal access of list or array indices until runtime.

Imagine having the two lists of `Integer` elements:

```
List<Integer> list_1 = new ListC<Integer>();
List<Integer> list_2 = new ListC<Integer>();
```

where the two lists are defined to be `[1,2,3,4,5]` and

`[1,2,3,4,5,6,7,8,9,10]` respetively, and suppose we want to perform a componentwise multiplication between these two lists, that is, we want to multiply corresponding indices of the two lists, with the following code:

```
for(int i=0; i < length(list_2); i++)
   list_1.get(i) * list_2.get(i);
```

Obviously, `list_1` and `list_2` have different lengths, with `list_1` having only 5 integers as opposed to 10 in `list_2`. While `i` is less than 5, this operation will function properly, but a runtime error will be thrown when we try to access an index `i` greater than the length of `list_1`. Despite Java's polymorphic type system, it does not have the capability of recognizing that we will have nothing to multiply to the second half of `list_2` at compile time.

4 The Dependent Lambda Calculus

The previous example of a list index runtime error provides the motivation for the implementation portion of this thesis. While most popular languages do not have type systems capable of catching this error during static analysis, dependent types aim to offer a sound method of solving this issue. By providing additional information about the program via expressive types, static analysis can know more about the program that would otherwise not be learned until runtime.

4.1 $\lambda\Pi$

Just as with other type theories, dependent type systems are based off of a theoretical lambda calculus with abstract syntax, typing, and evaluation rules. The dependently typed lambda calculus, $\lambda\Pi$, is more of an overhaul of the traditional lambda calculus than other theories, such as System F, that were merely extensions with additional syntax and slight variations on formal rules.

4.1.1 Syntax

As we did with the simply typed $\lambda_{\rightarrow}$, we will provide the abstract syntax of $\lambda\Pi$ along with type and evaluation semantics. As opposed to the type theories discussed so far, $\lambda\Pi$ mixes types into all other expressions. In other systems, we differentiate and only allow types to occur in specific locations

of expressions, as in annotations or type parameters in System F, but $\lambda\Pi$ allows types to occur wherever usual terms are allowed [1]. We will see how this feature becomes important as we delve deeper into $\lambda\Pi$.

$e, \rho :=$	$\star$	Type of Types (Star)
$\mid$	$\forall x : \rho.\rho'$	Dependent Function Space
$\mid$	x	Variable
$\mid$	c	Constant
$\mid$	$e\ e'$	Application
$\mid$	$\lambda x{:}\rho.e$	Lambda Abstraction
$v, \tau :=$	c_0	Constant Value
$\mid$	$\star$	Type of Types
$\mid$	$\forall x : \tau.\tau'$	Dependent Function Space
$\mid$	$\lambda x{:}\tau.e$	Lambda Abstraction

Figure 6: Abstract Syntax of $\lambda\Pi$

To make the abstract syntax more clear, we use e and ρ to refer to what we typically think of as terms and types respectively, but in this system, they are interchangeable and just used for syntactic clarity.

We also introduce the abstract syntax for values. Values can either be constant values in the language, $\star$, the type of types, a dependent function type, or a lambda abstraction. These values are the evaluated forms of the terms in the term syntax of $\lambda\Pi$.

We introduce new constructs to the syntax from the simply-typed lambda calculus. Here, $\star$, the 'type of types', is introduced as a term. Arrow types are subsumed by the newly introduced dependent function space, the dependent type of functions. With these new introductions, type terms coincide with all other terms.

Similarly, we intermingle types with values. Thus anywhere we can evaluate a portion of a program to a value, we can also evaluate to types. This allows types to depend on specific values, the core idea behind expressive dependent types that capture more errors than simple types.

4.1.2 Evaluation

While we did not discuss evaluation in simply-typed or polymorphic contexts, type-checking in $\lambda\Pi$ utilizes evaluation. We will see what this means shortly, but here we present evaluation rules and discuss their meanings.

$$\frac{}{\star \Downarrow \star} \qquad \frac{\rho \Downarrow \tau \qquad \rho' \Downarrow \tau'}{\forall x : \rho.\rho' \Downarrow \forall x : \tau.\tau'} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e \Downarrow \lambda x{:}\tau.e \qquad e[x \coloneqq e'] \Downarrow v}{e\ e' \Downarrow v} \qquad \frac{\rho \Downarrow \tau}{\lambda x{:}\rho.e \Downarrow \lambda x{:}\tau.e}$$

Figure 7: Evaluation Rules for $\lambda\Pi$ [1]

When we evaluate the $\star$ term, it simply evaluates to the $\star$ value.

When evaluating a dependent function space, we evaluate the domain ρ, and the codomain ρ' to evaluate the whole term.

Evaluating applications, we first evaluate the operator term to a lambda abstraction value. We then replace all x in the body of the lambda abstraction with e', and evaluate this substituted body to v, thus the result of the application is v. If the operator term does not evaluate to a lambda abstraction value, then this is an illegal application that will be caught during type checking before ever reaching evaluation.

Finally, to evaluate a lambda abstraction, we just evaluate the type annotation $\rho \Downarrow \tau$ and return the resulting lambda abstraction value substituting τ for ρ.

To provide a concrete example of evaluation in $\lambda\Pi$, consider working within the context Γ with integer constants, and a function $\lambda x{:}int.x^2$ bound to the variable f. Then we can evaluate the following expression as follows:

$$\frac{f \Downarrow \lambda x{:}int.x^2 \qquad \dfrac{8^2 \Downarrow 64}{x^2[x \coloneqq 8] \Downarrow 64}}{f\ 8 \Downarrow 64}$$

4.1.3 Type Checking

As was the case with $\lambda_{\rightarrow}$, the dependently typed lambda calculus associates a type semantics to its terms that allow us to implement a type checking algorithm for static program analysis. Unique to $\lambda\Pi$, you will see that some type rules involve evaluation of certain terms [1, 2]. This is because types may depend on specific values. As the clearest example, the type of an application may depend on the value of the input parameter,

thus typing involves evaluating the input, and type checking the type of the codomain of the dependent function space in a context, extended with the newly evaluated input.

$$\frac{}{\Gamma \vdash \star : \star}\ (\textsc{Star}) \qquad \frac{\Gamma \vdash \rho : \star \qquad \rho \Downarrow \tau \qquad \Gamma[x \coloneqq \tau] \vdash \rho' : \star}{\Gamma \vdash (\forall x : \rho.\rho') : \star}\ (\textsc{Pi})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash \rho : \star \qquad \rho \Downarrow \tau \qquad \Gamma[x \coloneqq \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\rho.e) : \forall x : \tau.\tau'}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash e : \forall x : \tau.\tau' \qquad \Gamma \vdash e' : \tau \qquad \tau[x \coloneqq e'] \Downarrow \tau''}{\Gamma \vdash e\ e' : \tau''}\ (\textsc{App})$$

Figure 8: Type Rules for $\lambda\Pi$ [1]

The Star term type checks as the type $\star$. Whenever a term type checks to $\star$, we say the it 'has kind $\star$', indicating that it is a type term. Essentially, this rule says that the type of types is itself, a type.

According to the (Pi) rule, a dependent function space has kind $\star$, i.e. is a type (specifically a function type). We first type check the domain term to make sure it has kind star, and then extend the context with the evaluated domain type bound to the variable x, and check that the codomain has kind $\star$ in this extended context. If this all succeeds, then the dependent function space is a valid type, and type checks as type $\star$.

When we type check a variable, we simply look up the type of the variable in the context.

For a lambda expression, we first check that the annotated type term ρ has kind $\star$. We then evaluate this type, and extend the context to include this type τ associated with the variable x when we type check the body.

As mentioned, application is an interesting case in $\lambda\Pi$. We first check that the type of the applicant term e is a dependent function type. If the operand term e' has the same type as the domain of the function space, then we substitute all occurences of the parameter x in the codomain type τ' with the operand term. If this evaluation after this substitution yields type τ'', then the resulting type of the whole application will be τ''.

4.1.4 Example Type Derivation in $\lambda\Pi$

To introduce the usage of these type rules, we look at the type derivation of a simple lambda abstraction $\lambda x{:}\star\,.\lambda y{:}x.y$.

$$\dfrac{\Gamma \vdash \star : \star \qquad \star \Downarrow \star \qquad \dfrac{\dfrac{\Gamma[x \coloneqq \star](x) = \star}{\Gamma[x \coloneqq \star] \vdash x : \star} \qquad x \Downarrow x \qquad \dfrac{\Gamma[x \coloneqq \star][y \coloneqq x](y) = x}{\Gamma[x \coloneqq \star][y \coloneqq x] \vdash y : x}}{\Gamma[x \coloneqq \star] \vdash \lambda y{:}x.y}\ (1)}{\Gamma \vdash (\lambda x{:}\star\,.\lambda y{:}x.y) : (\forall x : \star.\forall y : x.x)}$$

This is the polymorphic identity function in $\lambda\Pi$. Although it is quite similar to System F, notice that the type parameter x is abstracted with the same lower-case λ as ther next parameter. This is because types are treated the same way as other terms, whereas System F has separate rules for type abstraction and term abstraction. Verbally, this judgment states that for any type x passed in as a parameter, this will return a function that takes an input of type x, and returns an output of type x.

As a more complex example of these type rules in action, we look at the type derivation of the following expression:

$$((\lambda x{:}\star\,.\lambda y{:}x.y)\ int)\ 4$$

Here we apply the identity function to the type parameter int with the value 4 as the second input. To derive the type of the application, we have:

$$\dfrac{{}^1\Gamma \vdash (\lambda x{:}\star\,.\lambda y{:}x.y)\ int : (\forall x : int.int) \qquad \dfrac{\Gamma(4) = int}{\Gamma \vdash 4 : int} \qquad \dfrac{int}{int[x \coloneqq 4] \Downarrow int}}{\Gamma \vdash ((\lambda x{:}\star\,.\lambda y{:}x.y)\ int)\ 4 : int}$$

$$\dfrac{\text{Type Derivation (1)} \qquad \dfrac{\Gamma(int) = \star}{\Gamma \vdash int : \star} \qquad \dfrac{\forall y : int.int}{\forall y : x.x[x \coloneqq int] \Downarrow \forall x : int.int}}{{}^1\Gamma \vdash (\lambda x{:}\star\,.\lambda y{:}x.y)\ int : (\forall x : int.int)}$$

where we substitute the derivation for the polymorphic identity function in place of 'Type Derivation (1)'. Thus the resulting type of the expression is int.

4.2 Dependent Lists

Without more complex datatypes, it can be difficult to see the benefits, and even just the differences, of $\lambda\Pi$ from System F. Polymorphism can be achieved to the same effect as with the polymorphic lambda calculus, but

the intention is to gain additional capabilities through the more complex type system. To take full advantage of $\lambda\Pi$, we introduce dependent list types. We want them to be polymorphic, so that we can declare a list of integers, a list of strings, or even a list of types, and to be able to statically ensure that we do not violate the prescribed type of a list by trying to insert an element of an improper type. Additionally, we include the length of a list in its type, thus we the type $\tau list(x)$, where τ has kind $\star$ and x is an integer. In concrete instances, this allows us to construct objects of such types as $int\ list(3)$, or $\star\ list(5)$ (read int list of length 3 and star list of length 5, that is a list of types of length 5).

To allow for such constructs in the abstract syntax, we allow a term to be of the forms:

$stop[\rho]$	Nil of type ρ
$more[e, \rho]\ e'\ e''$	Cons of length e, type ρ, element e', and rest e''
$\rho\ list(e)$	List of type ρ, length e

in addition to the existing allowable syntactic constructs. We want to define type rules for lists such that we can derive types for expressions such as

```
more[2,int] 10 more[1,int] 20 stop[int]:int list(2)
```

and

```
more[2,⋆] int more[1,⋆] ∀x:bool.bool list(3) stop[⋆]:⋆ list(2)
```

(Removing the syntactic necessities for the dependent type system, these are just the lists `[10,20]` and `[int,∀x:bool.bool list(3)]` respectively).

4.2.1 Type Rules for Lists

When we type check lists, we want to check that the elements of a list are of the correct type, and that the length of the list corresponds to the length attribute of the type itself. We have two constructors for lists, so we will need two rules, one to type check a Nil or empty list, and one to type check a Cons (i.e. a non-empty list). We also need a type rule to assert that a list type term has kind $\star$ (i.e. is a valid type).

$$\frac{\Gamma \vdash \rho : \star \qquad \rho \Downarrow \tau}{\Gamma \vdash stop[\rho] : \tau\ list(0)}\ (\textsc{Nil})$$

$$\frac{\begin{array}{cccc} e \Downarrow v & \rho \Downarrow \tau & & \\ \Gamma \vdash e : int & \Gamma \vdash \rho : \star & \Gamma \vdash e' : \tau & \Gamma \vdash e'' : \tau\ list(v-1) \end{array}}{\Gamma \vdash more[e, \rho]\ e'\ e'' : \tau\ list(v)}\ (\textsc{Cons})$$

$$\frac{\Gamma \vdash \rho : \star \qquad \Gamma \vdash e : int}{\Gamma \vdash \rho\ list(e) : \star}\ (\textsc{List})$$

Figure 9: Type Rules for List Constructs in $\lambda\Pi$

To type check a Nil list, we first check that ρ has kind $\star$, and then evaluate ρ to τ to determine the type of the elements of the list.

Type checking a Cons cell is the most computationally expensive portion of type checking in our language. As shown in the type rule above, type checking involves evaluation of terms. We first ensure that the provided length term is of type *int*, and evaluate it to v. We then check that ρ is of kind $\star$, and evaluate it to the value τ. If the element e'' has type τ, and the rest, or tail, has type $\tau\ list(v-1)$, that is a list of length $v-1$ and elements of type τ, then the Cons type checks to type $\tau\ list(v)$.

To assert that a list type term has kind $\star$, we only need to check that the term ρ has kind $\star$, and that e has type *int*.

It is evident from these type rules that adding lists to $\lambda\Pi$ requires supporting an *int* type. We take for granted in this report that elements of this type behave as we expect, but must implement *int* before being able to support lists.

4.3 An example

While the type rules get more complex, dependent lists are the culmination of what we have discussed so far, so a small example can only help illuminate the benefits of this extension of $\lambda\Pi$. Given the list of *int* elements [2,4] (in our syntax written `more[2,int] 2 more[1,int] 4 stop[int]` with type $int\ list(2)$), we derive its type from our rules as follows:

$$\frac{\begin{array}{cccc} 2 \Downarrow 2 & int \Downarrow int & & \\ \Gamma \vdash 2 : int & \Gamma \vdash int : \star & \Gamma \vdash 2 : int & {}^{(1)}\Gamma \vdash more[1, int]\ 4\ stop[int] : int\ list(1) \end{array}}{\Gamma \vdash more[2, int]\ 2\ more[1, int]\ 4\ stop[int] : int\ list(2)}$$

We can apply the same type rule to derive the type of the tail in expression (1), and then apply the Nil type rule to derive the type of the end of the list. This displays that even for a simple, explicitly declared list of length 2, type checking is expensive, so why do we go through all of this effort? We can now incorporate dependent lists into functions to ensure additional type-safety that other languages are incapable of promising.

5 Implementation

So far we have only discussed theoretical calculi for language design, but the goal is to implement a language abiding by these rules for practical use. Accompanying this thesis is an OCaml implementation of $\lambda\Pi$, extended with dependent lists, and familiar language constructs such as if-statements, boolean, and integer values. We will discuss some design choices, and follow with examples to highlight the implemented type system in action.

5.1 Implementation Decisions

Much of my implementation work was directed by the paper, *A tutorial implementation of a depeendently typed lambda calculus*, presenting type and evaluation semantics and instructions for a Haskell implementation [1]. When working on my own implementation, portions of this paper depended upon more complex concepts or Haskell-specific features. My aim was to implement $\lambda\Pi$ in the most straightforward, general way possible, hence the title of an ML implementation, rather than OCaml.

5.1.1 Abstract Syntax

When implementing the language, a choice was made to maintain a first-order abstract syntax. For dependent function types and lambda abstractions, where the types of the codomain or body depend on a value, one could alternatively represent them as functions in the implementation language itself to automatically handle substitution [1]. For example, whereas I implemented the constructor for a dependent function space to accept a variable bound to a type for the domain, and a type for the codomain that relies on substitution, one could instead implement it to accept a domain type, and an OCaml function for the codomain, that given a value, returns a type. Although a higher-order abstract syntax may reduce overall complexity, and even may be a more natural way of implementation, the additional

abstraction obfuscates the underlying semantics of $\lambda\Pi$ more than I wanted to allow.

5.1.2 Bound and Free Variables

The tutorial implementation utilizes de Bruijn indices to maintain bound variables and their scopes throughout the intermediate representation of the language [1]. Again, though this may be the standard implementation practice, I chose to instead implement direct substitution of variables bound to symbols. To avoid name capture, I maintained a context containing information of types and values for both type checking and evaluation.

5.1.3 Term Equality

Because of the previous choices made in the tutorial paper, checking equality of terms became non-trivial. While it would be easy to assert that the *int* type is equal to the *int* type, but not equal to the *bool* type, the higher-order abstract syntax makes certain cases more difficult to compare. Because functions are used to define language constructs for dependent function spaces and lambda abstractions, we cannot simply ask if two OCaml functions are equal to determine the equality of the term itself. We would need to assert that the two OCaml functions produced the same output on every given input, a task that sounds too complicated to even attempt to address.

A technique called 'quotation' was used to handle this issue, effectively reverting higher-order abstract syntactic representations of values to first-order terms that can be syntactically checked for equality [1]. Once again, this did not uphold my expectations of clarity and simplicity in my own implementation, and fortunately by avoiding a higher-order abstract syntax altogether, I was able to implement a straightforward syntactic comparison to determine equality.

5.2 Demo

To test my implementation of the language and confirm it properly adheres to the semantics of $\lambda\Pi$, I implemented a read-eval-print-loop interpreter to demo some code written in my language. It is important to realize that this interpreter conflates type checking and evaluation of terms into a single process, and so appears to behave as a dynamically typed interactive language. Although these errors seem to occur only during program

execution, type checking and evaluation are actually implemented as separate processes, that in a compiler would be separated into pre-compile type checking that produces a compiled executable of the program if type checking is satisfied, and evaluation of this already statically analyzed executable program.

5.2.1 Identity example

To introduce the syntax of my implementation, we begin with a presentation of the polymorphic identity function we have previously shown the typing rules for in $\lambda\Pi$.

```
let id:pi x:∼.pi y:x.x =
       fn x:∼.fn y:x.y;;

id int 8;;                                    (*Evaluates to 8*)
id (pi x:∼.pi y:x.x) id;;          (*Evaluates to 'id' function*)
id int id;;                                  (*Fails to type check*)
```

Function declarations mimic the syntax of a lambda abstraction, and a `let` statement just allows us to bind the function to a variable name. If the type of the lambda abstraction does not match the annotated type of the variable name in the `let` expression, type checking fails. Also, note that in the concrete syntax, $\sim$ replaces $\star$ as the type of types symbol.

In the case of the polymorphic identity function, we annotate the function name with the type of the function itself, which we see is `pi x:∼.pi y:x.x` . In the abstract syntax of $\lambda\Pi$, this is equivalent to the dependent function space $\forall$`x:`$\star$.$\forall$`y:x.x`. Althought the type output by the application of `id` to a single parameter depends on the first parameter, this is an instance of types depending on types, not types depending on values. This is nearly equivalent to parametric polymorphism in System F, instead of dependent types.

5.2.2 `n_copies` Example

Now that the syntax of the language is slightly more understandable, we can write an `n_copies` function to create a list of n copies of an input x.

```
let n_copies:pi t:∼.pi n:int.pi x:t.t list(n) =
   fn t:∼.fn n:int.fn x:t.if n == 0 then stop[t]
       else more[n,t] x (n_copies t (n-1) x);;

n_copies int 3 0;;                                    (*[0,0,0]*)
n_copies ∼ 2 pi x:int.int;;      (*[pi x:int.int,pi x:int.int]*)
n_copies bool true 1;;                    (*Fails to type check*)
```

The `n_copies` function accepts three parameters, the first being the type, then the length, then the element to copy. If these are supplied in the wrong order, as in the last case, the application will fail to type check. Such an error is now caught during compilation, whereas our corresponding Python function did not catch errors until runtime.

The `n_copies` function is our first example of a truly dependent function space. The output type of the function is `t list(n)`. Not only does this type depend on the type parameter `t`, but it also depends on the value of `n` to which the function is applied, as the type of a list depends on its length. Indeed, the length of the list could be any arbitrary term that we need not know until compile time, but we can still ensure well-typedness [2].

5.2.3 Dependent Lists in Action

We finally reach the motivating example for dependent types, as we will write a componentwise multiplication function on two lists of integers, that requires them to be of the same length, or otherwise fails to type check.

```
let comp_mult:pi n:int.pi a:int list(n).pi b:int list(n).int list(n) =
      fn n:int.fn a:int list(n).fn b:int list(n).
          if n == 0 then stop[int]
          else more[n,int] ((head a) * (head b))
               (comp_mult (n-1) (tail a) (tail b));;

comp_mult 1 (more[1,int] 5 stop[int])
    (more[1,int] 5 stop[int]);;                               (*[25]*)
comp_mult 1
    (more[1,bool] true stop[bool])          (*Fails to type check*)
    (more[1,bool] false stop[bool]);;
comp_mult 1 (more[1,int] 7 stop[int])       (*Fails to type check*)
    (stop[int]);;
```

When we componentwise multiply two integer lists of length 1, the application passes the type checking process, so compilation would continue and we can expect the program to execute properly and return the list `[25]`. If we try to componentwise multiply lists on elements that are of non-integer type, then the type system catches a type error during compilation, just as we would expect in a statically typed language such as Java. Finally, we see the power of $\lambda\Pi$ as a type system. When we try to componentwise multiply two integer lists of different lengths, type checking fails, as we require both lists input to be of length 1.

6 Conclusion

These examples show how $\lambda\Pi$ effectively handles the same typed situations as less expressive type theories. Simply-typed and polymorphic terms remain well-typed in $\lambda\Pi$, and we have introduced more information to the type system to capture more understanding during static analysis. Dependent lists are a powerful extension of the dependently typed lambda calculus; they maintain the flexibility of polymorphic lists and express more about the state of the program than simple polymorphic lists by keeping track of the list's length.

Despite contributing to the computational cost of static analysis, $\lambda\Pi$ is reasonably practical to implement for general programming. Dependent types may not be the solution, or even incorporated in high-performance languages because of their complexity, but for a slight increase in compilation time, significant assurance can be gained that a well-typed program will execute as expected.

With dependent function spaces and dependent list types already incorporated in the core of my language, further extensions of dependent types should become easier to implement. While certain issues may not be captured by the dependent types discussed in this paper, dependent types can be extended to capture features such as array index bounds, or even to ensure a binary heap structure to a tree.

It is still true that certain issues remain inherently runtime errors. Specifically, dependent types cannot do much in the way of ensuring user input during execution satisfies the dependent type system, and therefore they are not capable of guaranteeing proper program execution. To counter this attack on dependent types, one only has to point out that such runtime errors would be runtime errors in any of the more popular, yet less strict, formal type systems at the basis of modern languages. Dependent types may miss

certain errors during static analysis, but catch all that would be caught in simple type systems, and additional errors that are beyond their scope.

7 Future Work

Hopefully the usefulness of dependent types is made clear from the previous discussions, but there is more work to be done in creating highly expressive type systems to ensure additional safety of program execution. It seems that the more expressive the type system, the more errors we are capable of catching during static analysis, so naturally we want to extend $\lambda\Pi$ further to reap its full benefits.

Though my current implementation introduces some extensions of $\lambda\Pi$ to incorporate assignments, conditionals, arithmetic, boolean logic, and dependent lists, much is lacking compared to what most immediately think of as a general programming language. Extending the core type system further to allow dependent abstract data types, other dependent data structures, and a core API strengthened by the theoretical foundation of $\lambda\Pi$ would be the next natural step in further developing this implementation.

Despite the expressive power of such a complex, but useful type theory, requiring types to contain such additional information inevitably clutters the syntax of a language. Not only do we have explicit type annotations, but these type annotations depend on evaluation of terms. Of course, it would be helpful to clean up the syntax of my implementation to improve ease of use, but this would also further dispel the notion that dependently typed languages are not suited for general programming. The highly expressive type systems based off of an encoding of predicate logic in language make for extremely powerful logic-based proof systems, but it is often thought that the complexity of dependent types is too high for common purposes. Hopefully a straightforward implementation and more succinct syntax will help discourage this belief.

Although $\lambda\Pi$ has defficiencies, the most notable of which is probably the computational cost of type checking a dependent language, it is important to recognize the usefulness we can gain from such additional security. While static typing will never even be universally accepted due to the restrictions it places on the programmer, we should still acknowledge the importance and practicality of more rigid formal systems thanks to their increased assurance of proper execution.

References

[1] Andres Löh , Conor McBride , Wouter Swierstra, *A Tutorial Implementation of a Dependently Typed Lambda Calculus*, Fundamenta Informaticae, v.102 n.2, p.177-207, April 2010

[2] Norell, Ulf. *Dependently typed programming in Agda.* Advanced Functional Programming. Springer Berlin Heidelberg, 2009. 230-266.

[3] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

Appendix

```
(*
 * file: ast.ml
 * author: Sam Baxter
 *
 * This file lays out the abstract syntax of the language.
 * A substitution function is implemented to handle
 * substitution of types and values in type checking and
 * the evaluation process.
 * A toString function is provided for the interactive display.
 *)

type variable =
  | String of string
  | Symbol of string * int
  | Dummy

type term =
  | Var of variable
  | Star
  | Pi of abstraction
  | Lambda of abstraction
  | App of term * term
  | Int of int
  | Bool of int
  | Ann of term * term
  | If of term * term * term
  | And of term * term
  | Or of term * term
  | Op of term * term list
  | Let of abstraction
  | IntType
  | BoolType
  | List of term * term
  | Prod of term list
  | BinaryOp of (term * term -> term)
  | UnaryOp of (term -> term)
  | Nil of term
  | Cons of term * term * term * term
  | IsNil of term
  | Head of term
  | Tail of term

and abstraction =
    variable * term * term

let fresh =
```

```
47|   let k = ref 0 in
      function
49|     |String x | Symbol(x, _) ->
          incr k;
51|         Symbol(x, !k)
      | Dummy ->
53|         incr k;
          Symbol("", !k)
55|
  let rec subst s = function
57|   | Var x ->
        (try List.assoc x s with Not_found -> Var x)
59|   | Star ->
        Star
61|   | Pi a ->
        Pi (subst_abs s a)
63|   | Lambda a ->
        Lambda (subst_abs s a)
65|   | App(e1, e2) ->
        App(subst s e1, subst s e2)
67|   | Int i ->
        Int i
69|   | Bool b ->
        Bool b
71|   | Ann(e1, e2) ->
        Ann(subst s e1, subst s e2)
73|   | If(e1, e2, e3) ->
        If(subst s e1, subst s e2, subst s e3)
75|   | And(e1, e2) ->
        And(subst s e1, subst s e2)
77|   | Or(e1, e2) ->
        Or(subst s e1, subst s e2)
79|   | Op(rator, rands) ->
        Op(rator, List.map (subst s) rands)
81|   | Let(x, typ, e) ->
        Let(x, subst s typ, subst s e)
83|   | IntType ->
        IntType
85|   | BoolType ->
        BoolType
87|   | List(typ, len) ->
        List(subst s typ, subst s len)
89|   | Nil e -> Nil (subst s e)
     | Cons(len, typ, el, rest) ->
91|     Cons(subst s len, subst s typ, subst s el, subst s rest)
     | IsNil e ->
93|     IsNil(subst s e)
     | Head e ->
95|     Head(subst s e)
```

```
  | Tail e ->
97      Tail(subst s e)


99
and subst_abs s (x, t, e) =
101   let x' = fresh x in
    (x', subst s t, subst ((x, Var x')::s) e)
103
let makeString s = String s
105
let rec toString = function
107   | Var x -> toStringVar x
    | Star -> "~"
109   | Pi a -> "Pi" ^ (toStringAbs a)
    | Lambda a -> "Lambda" ^ (toStringAbs a)
111   | App(e, e') -> "App(" ^ (toString e) ^ ", " ^ (toString e') ^
      ")"
    | Int i -> string_of_int i
113   | Bool i -> if i = 0 then "false" else "true"
    | Ann(e, t) -> toString e
115   | If(e1, e2, e3) ->
        "if " ^ (toString e1) ^ " then " ^ (toString e2) ^ " else
     " ^ (toString e3)
117   | And(e1, e2) ->
        "&(" ^ (toString e1) ^ ", " ^ (toString e2) ^ ")"
119   | Or(e1, e2) ->
        "||(" ^ (toString e1) ^ ", " ^ (toString e2) ^ ")"
121   | Op(rator, rands) -> (toString rator) ^ "(" ^ (toStringList
      rands) ^ ")"
    | Let a -> "Let" ^ (toStringAbs a)
123   | IntType -> "int"
    | BoolType -> "bool"
125   | List(typ, len) -> (toString(typ)) ^ " list(" ^ (toString len
      ) ^ ")"
    | Prod x -> toStringTuple x
127   | BinaryOp f -> "bi-op"
    | UnaryOp f -> "u-op"
129   | Nil e -> "[]:" ^ (toString e)
    | Cons(len, typ, el, rest) -> "more[" ^ (toString len) ^ "," ^
       (toString typ) ^ "] " ^ (toString el) ^ " " ^ (toString rest
      )
131   | IsNil e -> "is_nil(" ^ (toString e) ^ ")"
    | Head e -> "head(" ^ (toString e) ^ ")"
133   | Tail e -> "tail(" ^ (toString e) ^ ")"

135 and toStringTuple = function
    | [] -> ""
137   | [a] -> toString a
    | x::xs -> (toString x) ^ " * " ^ (toStringTuple xs)
```

```
139
and toStringList = function
141   | [] -> ""
      | [only] -> toString only
143   | x::xs -> (toString x) ^ ", " ^ (toStringList xs)

145 and toStringVar = function
      | String s -> s
147   | Symbol (s, i) -> s

149 and toStringAbs (x, t, e) = "(" ^ (toStringVar x) ^ ", " ^ (
      toString t) ^ ", " ^ (toString e) ^ ")"
```

Listing 1: ./../../Final/ast.ml

```
1 (*
  * file: environment.ml
3 * author: Sam Baxter
  *
5 * This file contains the definition of contexts.
  * Functions are included to extend and look up associations
7 * in the context by type checking and evaluation functions.
  *)
9
type context = (Ast.variable * (Ast.term * Ast.term option))
    list
11
let lookup_typ x ctx = fst (List.assoc x ctx)
13
let lookup_value x ctx = snd (List.assoc x ctx)
15
let extend x t ?value ctx = ctx := (x, (t, value)) :: !ctx
```

Listing 2: ./../../Final/environment.ml

```
(*
2 * file: basis.ml
  * author: Sam Baxter
4 *
  * This file contains the code for building both the static and
     dynamic basis.
6 * The function makeBasis will construct a static basis when
     applied to the
  * list Basis.primOpTypes.  The same function will construct a
     dynamic
8 * basis when applied to the list implementationsOfPrimitives.
      These bases
  * are constructed in the preamble to the top-level read-eval-
     print loop in
```

```
 * interpreter.ml  The apply functions can be used to actually
    put a type checker for (or
 * implementation of) a primitive to use.
 *
 * To extend the basis with new primitives, the list primNames
    must be
 * extended with an appropriate identifier.
 *)

open Environment;;

(*
 * This is the master list of names of primitive operators.
 *
 * NB: THESE NAMES ARE LAYED OUT IN A FIXED ORDER!
 *)
let primOpNames = ["+"; "-"; "*"; "/"; "%"; "**"; "<"; "<="; "==
    "; "<>"; ">"; ">="; "not"];;

let rec zip = function
  | [], [] -> []
  | x::xs, y::ys -> (x, y)::(zip(xs, ys))

let makeBasis values =
  let primOpNames' = List.map Ast.makeString primOpNames in
  let keyValuePairs = zip(primOpNames', values) in
  let insert map (key,value) = (key, value)::map
  in
    List.fold_left insert [] keyValuePairs;;

module Interpreter =
  struct
    let applyBinary = function
  (operation,[value1;value2]) -> operation(value1,value2)
      | _ -> raise (Failure "Cannot happen.");;

    let applyUnary = function
  (operation,[value]) -> operation(value)
      | _ -> raise (Failure "Cannot happen.");;

    (*
     * The implementation of primitive operations. Note that the
     order of
     * these together with the unarys must match up with the
    order of the
     * operator names in op.ml.
     *)
    let intCrossInt2Int =
```

```
    Ast.Pi(Ast.makeString "qqqqqqq", (Ast.Prod [Ast.IntType;
  Ast.IntType]), Ast.IntType);;

  let intCrossInt2Bool =
    Ast.Pi(Ast.makeString "qqqqqqq", (Ast.Prod [Ast.IntType;
  Ast.IntType]), Ast.BoolType);;

  let bool2Bool =
    Ast.Pi(Ast.makeString "qqqqqqq", Ast.Prod([Ast.BoolType]),
   Ast.BoolType);;

  let operatorTypes =
    [
intCrossInt2Int;   (* + *)
intCrossInt2Int;   (* - *)
intCrossInt2Int;   (* * *)
intCrossInt2Int;   (* / *)
intCrossInt2Int;   (* % *)
intCrossInt2Int;   (* ** *)
intCrossInt2Bool;  (* < *)
intCrossInt2Bool;  (* <= *)
intCrossInt2Bool;  (* == *)
intCrossInt2Bool;  (* <> *)
intCrossInt2Bool;  (* >= *)
intCrossInt2Bool;  (* > *)
bool2Bool          (* not *)
    ];;

  let staticBasis = makeBasis operatorTypes;;

  let binaryPrePrimOps =
    [
      (function
        | (Ast.Int(v1),                    (* + *)
           Ast.Int(v2)) ->
            Ast.Int(v1+v2)
        | (a, b) ->
            Ast.Op(Ast.Var(Ast.String("+")),[a;b]));

(function
        | (Ast.Int(v1),
     Ast.Int(v2)) ->                (* - *)
      Ast.Int (v1 - v2)
        | (a,b) ->
            Ast.Op(Ast.Var(Ast.String("-")), [a;b]));

(fun (Ast.Int(v1),
      Ast.Int(v2)) ->                (* * *)
  Ast.Int (v1 * v2));
```

```
100   (fun (Ast.Int(v1),
            Ast.Int(v2)) ->                (* / *)
102     Ast.Int (v1 / v2));

104   (fun (Ast.Int(v1),
            Ast.Int(v2)) ->                (* % *)
106     Ast.Int (v1 mod v2));

108   (fun (Ast.Int(v1),
            Ast.Int(v2)) ->                (* ** *)
110     let v1' = float_of_int v1 in
              let v2' = float_of_int v2 in
112           Ast.Int(int_of_float(v1' ** v2')));

114   (fun (Ast.Int(v1),
            Ast.Int(v2)) ->                (* < *)
116     Ast.Bool(if v1 < v2 then 1 else 0));

118   (fun (Ast.Int(v1),
            Ast.Int(v2)) ->                (* <= *)
120     Ast.Bool(if v1 <= v2 then 1 else 0));

122   (function
            |(Ast.Int(v1),
124       Ast.Int(v2)) ->                (* == *)
            Ast.Bool(if v1 = v2 then 1 else 0)
126           |(a,b) ->
                Ast.Op(Ast.Var(Ast.String("==")), [a;b]));
128
    (function
130           | (Ast.Int(v1),
         Ast.Int(v2)) ->                (* <> *)
132        Ast.Bool(if v1 <> v2 then 1 else 0)
             | (a,b) ->
134             Ast.Op(Ast.Var(Ast.String"<>"), [a;b]));

136   (function
            | (Ast.Int(v1),
138       Ast.Int(v2)) ->                (* >= *)
            Ast.Bool(if v1 >= v2 then 1 else 0)
140           | (a, b) ->
                Ast.Op(Ast.Var(Ast.String(">=")), [a;b]));
142
    (function
144           | (Ast.Int(v1),
         Ast.Int(v2)) ->                (* > *)
146        Ast.Bool(if v1 > v2 then 1 else 0)
             | (a,b) ->
```

```
148                 Ast.Op(Ast.Var(Ast.String(">")), [a;b]));

150         ];;

152         (*
             * Coerce the implementations of binary primitives to be
        Ast.letues.
154         *)
            let binaryPrimOps = List.map (fun x -> Ast.BinaryOp x)
        binaryPrePrimOps;;
156
            (*
158          *  The unary primtives.
             *)
160         let unaryPrePrimOps =
    [
162             (function
                  |(Ast.Bool(v)) ->              (* not *)
164                   Ast.Bool(if v = 1 then 0 else 1)
                  |(a) ->
166                   Ast.Op(Ast.Var(Ast.String("not")), [a]))
    ];;
168
            (*
170         * Coerce the implementations of unary primitives to be
        Ast.values.
            *)
172         let unaryPrimOps = List.map (fun x -> Ast.UnaryOp x)
        unaryPrePrimOps;;

174         (*
             * Make the dynamic basis for export to the interpreter.
176          *)
            let dynamicBasis = makeBasis (binaryPrimOps @ unaryPrimOps
        );;
178     end;;
```

Listing 3: ./../../Final/basis.ml

```
    (*
2    * file: staticsemantics.ml
     * author: Sam Baxter
4    *
     * This file contains a normalize function, which in effect acts
         as
6    * an evaluation function on the abstract syntax. The infer
        funtion
     * infers types of terms and checks their validity according to
        the
```

```
 * formal rules of the dependent type system. The equal function
 * compares the equality of terms/types.
 *)

open Ast
open Environment

let rec normalize env = function
  | Var x ->
      (match
          (try lookup_value x !env
           with Not_found -> raise (Failure "unknow identifier\n
    "))
        with
          | None -> (Var x, env)
          | Some e -> (fst(normalize env e), env))
  | Star ->
      (Star, env)
  | Pi a ->
      (Pi (normalize_abs env a), env)
  | Lambda a ->
      (Lambda (normalize_abs env a), env)
  | App(e1, e2) ->
      let e2' = fst (normalize env e2) in
      (match fst (normalize env e1) with
        | Lambda (x, _, e1') ->
            (fst (normalize env (subst [(x, e2')] e1')), env)
        | e1 ->
            (App(e1, e2)), env)
  | Int i ->
      (Int i, env)
  | Bool b ->
      (Bool b, env)
  | Ann(e1, e2) ->
      (Ann(fst (normalize env e1),fst (normalize env e2)), env)
  | If(e1, e2, e3) ->
      (match fst (normalize env e1) with
        | Bool 1 -> normalize env e2
        | Bool 0 -> normalize env e3)
  | And(e1, e2) ->
      (match fst (normalize env e1) with
        | Bool 0 as t -> (t, env)
        | Bool 1 -> normalize env e2)
  | Or(e1, e2) ->
      (match fst (normalize env e1) with
        | Bool 1 as t -> (t, env)
        | Bool 0 -> normalize env e2)
  | Op(rator, rands) ->
      let rator' = fst (normalize env rator) in
```

```
56        (match rator', rands with
            | BinaryOp f, [a;b] ->
58              (f (fst (normalize env a), fst (normalize env b)),
      env)
            | UnaryOp f, [a] ->
60              (f (fst (normalize env a))), env)
    | Prod x ->
62        (Prod (List.map fst (List.map (normalize env) x)), env)
    | Let (x, t, e) ->
64        let t' = fst (normalize env t) in
          let e' = fst (normalize env e) in
66        extend x t' ~value:e' env;
          (Let (x, t', e'), env)
68  | IntType ->
          (IntType, env)
70  | BoolType ->
          (BoolType, env)
72  | BinaryOp f as x -> (x, env)
    | UnaryOp f as x -> (x, env)
74  | List(typ, len) ->
          (List(fst (normalize env typ), fst (normalize env len)),
      env)
76  | Nil e -> (Nil (fst (normalize env e)), env)
    | Cons(len, typ, el, rest) ->
78        (Cons(fst (normalize env len), fst (normalize env typ),
      fst (normalize env el), fst (normalize env rest)), env)
    | IsNil e ->
80        (match fst (normalize env e) with
            | Nil a as t-> (Bool(1), env)
82          | Cons(_, _, _, _) -> (Bool(0), env)
            | _ -> raise (Failure "Input cannot normalize\n"))
84  | Head e ->
          (match fst (normalize env e) with
86          | Cons(_, _, e, _) -> (e, env)
            | _ -> raise (Failure "Cannot normalize head of anything
       other than non-empty list\n"))
88  | Tail e ->
          (match fst (normalize env e) with
90          | Cons(_, _, _, e) -> (e, env)
            | Nil e -> (Nil e, env)
92          | _ -> raise (Failure "Cannot normalize tail of anything
       other than a list\n"))
    | _ -> raise (Failure "Input cannot normalize\n")
94
and normalize_abs env (x, t, e) =
96  let t' = fst (normalize env t) in
    (x, t', e)
98
let rec all_true l = (match l with
```

```
100|   | [] -> true
       | [x] -> x
102|   | x::xs -> x && all_true xs)

104| let rec apply_list fs ls =
       (match fs, ls with
106|     | [], [] -> []
       | [f], [x] -> [f x]
108|     | x::xs, y::ys -> (x y)::(apply_list xs ys))

110| let equal env e1 e2 =
       let rec equal' e1 e2 = (match e1, e2 with
112|     | Var x1, Var x2 -> x1 = x2
       | App(d1, d2), App(f1, f2) -> equal' d1 f1 && equal' d2 f2
114|     | Star, Star -> true
       | Pi a1, Pi a2 -> equal_abs a1 a2
116|     | Lambda a1, Lambda a2 -> equal_abs a1 a2
       | Int i, Int j -> i = j
118|     | Bool b, Bool b' -> b = b'
       | Ann(d1, d2), Ann(f1, f2) ->
120|       equal' d1 f1 && equal' d2 f2
       | Op(r, rands), Op(r', rands') ->
122|       equal' r r' && all_true (apply_list (List.map equal'
    rands) rands')
       | Let a1, Let a2 ->
124|       equal_abs a1 a2
       | IntType, IntType -> true
126|     | BoolType, BoolType -> true
       | Prod a, Prod b ->
128|       (match a, b with
           | [], [] -> true
130|         | [x], [y] -> equal' x y
           | x::xs, y::ys -> equal' (Prod xs) (Prod ys))
132|     | List(a, b), List(x, y) ->
         equal' a x && equal' b y
134|     | Nil a, Nil b ->
         equal' a b
136|     | Cons(a1, b1, c1, d1), Cons(a2, b2, c2, d2) ->
         equal' a1 a2 && equal' b1 b2 && equal' c1 c2 && equal'
    d1 d2
138|     | IsNil a, IsNil b ->
         equal' a b
140|     | Head a, Head b ->
         equal' a b
142|     | Tail a, Tail b ->
         equal' a b
144|     | _, _ -> false)
     and equal_abs (x, t, e1) (x', t', e2) =
146|     let z = Var (fresh x) in
```

```
    equal' t t' && (equal' (subst [(x, z)] e1) (subst [(x', z)]
    e2))
148  in
   equal' (fst (normalize env e1)) (fst (normalize env e2))
150
let rec infer env = function
152  | Var x ->
       (try lookup_typ x !env
154     with Not_found -> raise (Failure "unknown identifier\n"))
   | Star -> Star
156  | Pi (x, t, e) ->
       let t' = infer env t in
158    let temp = !env in
       extend x t env;
160    let e' = infer env e in
       env := temp;
162    (match t', e' with
           Star, Star -> Star
164      | _, _ -> raise (Failure "invalid type in dependent
    function space\n"))
   | Lambda (x, t, e) ->
166    let t' = infer env t in
       let temp = !env in
168    extend x t env;
       let e' =
170      (try infer env e
         with Failure s ->
172        env := temp;
           raise (Failure ("Input does not type-check\n" ^ s)))
    in
174    env := temp;
       Pi (x, t, e')
176  | App(e1, e2) ->
       let (x, s, t) = infer_pi env e1 in
178    let e2' = infer env e2 in
       check_equal env s e2';
180    subst [(x, e2)] t
   | Int i -> IntType
182  | Bool b -> BoolType
   | Ann(e1, e2) ->
184    let t = infer env e1 in
       check_equal env t e2;
186    t
   | If(e1, e2, e3) ->
188    (try check_equal env (infer env e1) (BoolType);
           check_equal env (infer env e2) (infer env e3);
190        infer env e2
       with Failure s ->
192      check_equal env (infer env e1) (BoolType);
```

```
        (match (infer env e2), (infer env e3) with
194       | List(t, a), List(t', b) ->
              check_equal env t t';
196           List(t', b)
          | List(_, Int i), List(_, Int j) -> raise (Failure "If
     statement on lists does not type-check\n")
198       | _, _ -> raise (Failure ("If statement does not type-
    check\n" ^ (toString (infer env e2)) ^ " <> " ^ (toString (
    infer env e3))))))
  | And(e1, e2) ->
200     let e1' = infer env e1 in
        let e2' = infer env e2 in
202     check_equal env e1' e2';
        check_equal env e1' BoolType;
204     BoolType
  | Or(e1, e2) ->
206     let [e1';e2'] = List.map (infer env) [e1;e2] in
        check_equal env e1' e2';
208     check_equal env e1' BoolType;
        BoolType
210 | Op(rator, rands) ->
        let (x, Prod(s), t) = infer_pi env rator in
212     let e = List.map (infer env) rands in
        apply_list (List.map (check_equal env) s) e;
214     subst [(x, Prod(rands))] t
  | Prod x -> Prod (List.map (infer env) x)
216 | Let(x, typ, e) ->
        let temp = !env in
218     extend x typ env;
        let t = infer env e in
220     (try check_equal env t typ
         with Failure s ->
222        env := temp;
           raise (Failure ("Let binding does not type-check\n" ^ (
    s))));
224     env := temp;
        t
226 | IntType ->
        Star
228 | BoolType ->
        Star
230 | List(typ, len) ->
        (match infer env typ, infer env len with
232       | Star, IntType -> Star
          | _ -> raise (Failure "Input does not type-check as list
    \n"))
234 | Nil e ->
        let t = fst (normalize env e) in
236     check_equal env (infer env t) Star;
```

```
      List(t, Int 0)
  | Cons(len, typ, el, rest) ->
      let len' = fst (normalize env len) in
      let typ' = fst (normalize env typ) in
      check_equal env (infer env typ') Star;
      check_equal env (infer env len') IntType;
      let el' = infer env el in
      check_equal env typ' el';
      (match len', infer env rest with
        | Int i, List(t, Int j) ->
            check_equal env t typ';
            (try assert (i = j+1)
             with Assert_failure s -> raise (Failure "List
    lengths do not type-check\n"));
            List(typ', len')
        | _, List(t, _) ->
            List(typ', len')
        | _ -> raise (Failure "Cons does not type-check\n"))
  | IsNil e ->
      (match infer env e with
        | List(_, Int i) -> BoolType
        | _ -> raise (Failure "Input does not type-check\n"))
  | Head e ->
      (match infer env e with
        | List(t, _) ->
            t
        | _ -> raise (Failure "Head does not type-check\n"))
  | Tail e ->
      (match infer env e with
        | List(t, Int i) as t'->
            if i = 0 then t'
            else List(t, Int (i-1))
        | List(t, a) ->
            List(t, Op(Var(String("-")), [a; Int 1]))
        | _ -> raise (Failure "Tail does not type-check\n"))
  | _ ->
      raise (Failure "General input does not type-check\n")


and infer_pi env e =
  let t = infer env e in
  (match fst (normalize env t) with
    | Pi a -> a
    | _ -> raise (Failure "dependent function space expected\n")
   )

and check_equal env x y =
```

```
284   if not (equal env x y) then raise (Failure ("Expressions are
       not equivalent\n" ^ (Ast.toString x) ^ " <> " ^ (Ast.toString
       y)))
```

Listing 4: ./../../Final/staticsemantics.ml

```
(* file: lexer.mll *)
(* Lexical analyzer returns one of the tokens:
   the token NUM of integer,
   operators (PLUS, MINUS, MULTIPLY, DIVIDE, CARET),
   or EOF.  It skips all blanks and tabs, unknown characters. *)
{
  open Parser (* Assumes the parser file is "parser.mly". *)
}
let digit = ['0'-'9']
let word = ['a'-'z''A'-'Z']
rule token = parse
  | [' ' '\t' '\n']{ token lexbuf }
  | ','      { COMMA }
  | digit+
  | "." digit+
  | digit+ "." digit* as num
      { NUM (int_of_string num) }
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { MULTIPLY }
  | '/'      { DIVIDE }
  | '%'      { MOD }
  | ':'      { COLON }
  | ';'      { SEMI }
  | "->"     { ARROW }
  | "fn"     { FUN }
  | "pi"     { PI }
  | "~"      { STAR }
  | '.'      { DOT }
  | "int"    { INTTYPE }
  | "bool"   { BOOLTYPE }
  | "list"   { LIST }
  | "stop"   { NIL }
  | "more"   { CONS }
  | "is_nil"{ ISNIL }
  | "head"   { HEAD }
  | "tail"   { TAIL }
  | "**"     { CARET }
  | "<"      { LT }
  | "<="     { LE }
  | "="      { EQ }
  | "=="     { CMPEQ }
  | "<>"     { NE }
```

```
44    | "!="      { NEALT }
      | ">="      { GE }
46    | ">"       { GT }
      | "true"    { TRUE }
48    | "false"   { FALSE }
      | "let"     { LET }
50    | "in"      { IN }
      | "if"      { IF }
52    | "else"    { ELSE }
      | "then"    { THEN }
54    | "and"     { AND }
      | "or"      { OR }
56    | "not"     { NOT }
      | word+ as string { ID string }
58    | '('       { LPAREN }
      | ')'       { RPAREN }
60    | '['       { LBRACKET }
      | ']'       { RBRACKET }
62    | _         { token lexbuf }
      | eof       { EOF }
```

Listing 5: ./../../Final/lexer.mll

```
1  /* file: parser.mly */

3  %{
     open Printf
5    let toId s = Ast.Var(Ast.makeString s)
   %}
7
   %token EOF
9  %token COMMA
   %token NIL CONS ISNIL HEAD TAIL LIST
11 %token <string> ID
   %token LPAREN RPAREN LBRACKET RBRACKET
13 %token <int> NUM
   %token LET IN COLON SEMI
15 %token PLUS MINUS MULTIPLY DIVIDE MOD CARET
   %token TRUE FALSE IF THEN ELSE AND OR NOT EQ
17 %token LT LE CMPEQ NE NEALT GE GT
   %token ARROW INTTYPE BOOLTYPE FUN PI DOT STAR
19
   %nonassoc CONS NIL ISNIL HEAD TAIL
21 %nonassoc COLON SEMI FUN PI
   %nonassoc IF THEN ELSE
23 %right DOT
   %left LET
25 %right EQ
   %right IN
```

```
%right ARROW
%left OR
%left AND
%left LT LE CMPEQ NE NEALT GE GT
%left PLUS MINUS
%left MULTIPLY DIVIDE MOD
%right CARET
%right COMMA
%nonassoc NOT TRUE FALSE LPAREN LBRACKET

%start input
%type <Ast.term> input

%% /* Grammar rules and actions follow */

input:
 exp SEMI SEMI EOF     { $1 }
;

exp:
| exp term             { Ast.App($1, $2) }
| term                 { $1 }
;

term:       NUM        { Ast.Int($1) }
| TRUE                 { Ast.Bool(1) }
| FALSE                { Ast.Bool(0) }
| FUN ID COLON tyterm DOT exp { Ast.Lambda(Ast.String($2), $4,
    $6) }
| tyterm               { $1 }
| ID                   { Ast.Var(Ast.String($1)) }
| exp COLON exp        { Ast.Ann($1, $3) }
| list                 { $1 }
| ISNIL exp            { Ast.IsNil($2) }
| HEAD exp             { Ast.Head ($2) }
| TAIL exp             { Ast.Tail ($2) }
| exp PLUS exp         { let id = toId("+") in Ast.Op(id,[$1; $3
    ]) }
| exp MINUS exp        { let id = toId("-") in Ast.Op(id,[$1; $3
    ]) }
| exp MULTIPLY exp     { let id = toId("*") in Ast.Op(id,[$1; $3
    ]) }
| exp DIVIDE exp       { let id = toId("/") in Ast.Op(id,[$1; $3
    ]) }
| exp CARET exp        { let id = toId("**") in Ast.Op(id,[$1; $3
    ]) }
| exp MOD exp          { let id = toId("%") in Ast.Op(id,[$1; $3
    ]) }
```

```
  | exp LT exp            { let id = toId("<") in Ast.Op(id,[$1; $3
      ]) }
69| exp LE exp            { let id = toId("<=") in Ast.Op(id,[$1; $3
      ]) }
  | exp CMPEQ exp         { let id = toId("==") in Ast.Op(id,[$1; $3
      ]) }
71| exp NE exp            { let id = toId("<>") in Ast.Op(id,[$1; $3
      ]) }
  | exp NEALT exp         { let id = toId("<>") in Ast.Op(id,[$1; $3
      ]) }
73| exp GT exp            { let id = toId(">") in Ast.Op(id,[$1; $3
      ]) }
  | exp GE exp            { let id = toId(">=") in Ast.Op(id,[$1; $3
      ]) }
75| NOT exp               { let id = toId("not") in Ast.Op(id,[$2])
      }
  | MINUS exp             { let id = toId("-") in Ast.Op(id,[Ast.Int
      (0); $2]) }
77| LPAREN exp RPAREN     { $2 }
  | IF exp THEN exp ELSE exp { Ast.If($2,$4,$6) }
79| exp AND exp           { Ast.And($1, $3) }
  | exp OR exp            { Ast.Or($1, $3) }
81| LET ID COLON tyterm EQ exp         { Ast.Let(Ast.String($2), $4
      , $6) }
  ;
83
  list:
85  NIL LBRACKET tyterm RBRACKET { Ast.Nil($3) }
  | CONS LBRACKET term COMMA tyterm RBRACKET exp term { Ast.Cons(
      $3, $5, $7, $8) }
87
  tyterm:
89  INTTYPE               { Ast.IntType }
  | BOOLTYPE              { Ast.BoolType }
91| PI ID COLON tyterm DOT tyterm  { Ast.Pi(Ast.String($2), $4, $6
      ) }
  | LPAREN tyterm RPAREN    { $2 }
93| ID                    { Ast.Var(Ast.String($1)) }
  | STAR                  { Ast.Star }
95| tyterm LIST LPAREN term RPAREN {Ast.List($1, $4) }
  ;
97
  %%
```

Listing 6: ./../../Final/parser.mly

```
2 module type INTERPRETER =
  sig
```

```
  open Ast
  open Environment
  val interpreter : Environment.context ref -> unit
  val makeContext : ('a * 'b) list -> ('c * 'd) list -> ('a * ('
    b * 'd option)) list
  val ctx : Environment.context ref
end;;

module Interpreter : INTERPRETER =
struct
  open Basis;;
  open Staticsemantics;;
  open Environment;;

  let stBasis = Basis.Interpreter.staticBasis;;
  let dyBasis = Basis.Interpreter.dynamicBasis;;

  let parseInput() =
    let inch = input_line stdin in
    let lexbuf = Lexing.from_string inch in
    let ast = Parser.input Lexer.token lexbuf in
    ast;;

  let rec interpreter context : unit =
    (
      output_string stdout ("\n>>> ");
      flush stdout;

      try
        (let ast = parseInput()
         in (try
               (
                 let e = Staticsemantics.infer context ast in
                 let t, ctx' = Staticsemantics.normalize context
    ast in
                 (
                   output_string stdout (Ast.toString t);
                   output_string stdout (":");
                   output_string stdout (Ast.toString e);
                   output_string stdout "\n";
                   flush stdout;
                   interpreter ctx'
                 )
               )
           with Failure s ->
             (
               output_string stdout s;
               flush stdout;
               interpreter context
```

```
                )
52            )
            )
54        with
              Parsing.Parse_error ->
56              (
                output_string stdout "Input string does not parse
    ...\n";
58              flush stdout;
                interpreter context
60              )
          | Failure s ->
62              (
                output_string stdout "Input does not type-check
    ...\n";
64              flush stdout;
                interpreter context
66              )
    );;
68  let rec makeContext x y = (match x, y with
    | [], [] -> []
70  | (a, b)::s, (c, d)::t -> (a, (b, Some d))::(makeContext s t
    ));;

72  let ctx = ref (makeContext stBasis dyBasis);;

74  let _ = interpreter ctx;;
end;;
```

Listing 7: ./../../Final/interpreter.ml