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 di erent theories on which to base these language designs, and this decision carries signi cant in uence over what a programming lanuage allows, or considers an error.

1.2.1 Syntax

The simply typed lambda calculus, $_{!}$, is the theoretical basis for typed, functional programming languages [3]. This is typically the rst formal system one learns about when studying type theory, and most typed languages handle type checking similarly to $_{!}$, so it makes sense for us to rst look at this calculus before considering more complex systems. $_{!}$ 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 or a function between types.

e :=		x	Variable
	j	c	Constant
	j	x: :e	Lambda Abstraction
	j	e e	Application
:=			Base Type
	j	!	Function Type

Figure 1: Abstract Syntax of $_{!}$

As an example, we can derive the expression

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

by following the steps

$$e\ 7 \to_{!}\ e\ e\ 7 \to_{!}\ (\ x{:}\ {:}e)\ e\ 7 \to_{!}\ (\ x{:}int{:}e)\ e\ 7 \to_{!}\ (\ x{:}int{:}x)\ e\ 7 \to_{!}\ (\ x{:}int{:}x)\ 7$$

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

Assuming we are within a context 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:

$$\dfrac{\dfrac{\dfrac{[x := int](x) = int}{[x := int] \vdash x : int}\,(\text{Var})}{\vdash (x{:}int{:}x) : int \to int}\,(\text{Lam}) \qquad \dfrac{(7) = int}{\vdash 7 : int}\,(\text{Const})}{\vdash (x{:}int{:}x)\ 7 : int}\,(\text{App})$$

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 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 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

$$\Lambda : e$$

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

$$\forall : \tau$$

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 $_{!}$. If we have a language with *int* and *String* as base types based o of the type rules of $_{!}$, we must write separate identity functions $x{:}int{:}x$ and $x{:}String{:}x$, with types $int ! int$ and $String ! String$ respectively, to operate on expressions of the two di erent base types. Although these are perfectly valid in $_{!}$, this creates redundant code that could be signi cantly condensed with a more powerful system, as the two functions perform equivalent functions.

In System F, we write the polymorphic identity function as

$$: x: :x$$

This allows us to abstract over the type of the input, and we supply the desired type as the rst parameter when we apply the function. That is, if we desire the identity function on objects of type *int*, we simply apply $: x: :x$ to *int*, and replace all 's in the body of the with *int*, yielding $x{:}int{:}x$. 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:

$$\frac{` e :}{` :e : 8 :} \text{(T-Lam)} \qquad \frac{` e : 8 : {}^{0} \quad ` : \text{Type}}{` e : {}^{0}[:=]} \text{(T-App)}$$

Figure 3: Extended Type Rules for System F

where ${}^{0}[:=]$ in the (T-App) rule means that we replace all occurrences of in ^{0} with , and the assertion ` : Type means that is a valid type in context . Thus we can provide a type judgement for the polymorphic identity function as follows:

$$\dfrac{\dfrac{\dfrac{[x :=](x) =}{[x :=] ` x :}\text{(Var)}}{` (x: :x) : !}\text{(Lam)}}{` (: x: :x) : 8 : !}\text{(T-Lam)}$$

and conclude that the polymorphic identity function in System F has type $8 : ! $.

As we see in this example, System F allows programmers much more exibility and power to write reusable, polymorphic code. This lies at the basis of polymorphic data structures, map, and fold functions that make code e cient 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 bene t to programmers by catching many mistakes before a program is ran, preventing runtime crashes. But even in languages based o 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 bene cial, 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 o of the dependently typed lambda calculus, , developed from Per Martin Lof's

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

This straightforward implementation of a List class is a syntactically nicer version of System F, where `Item` takes the place of all 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 o of the formal System F, and adheres to its theoretical type rules.

3.3 De ciencies

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 de ned to be `[1,2,3,4,5]` and

4.1.2 Evaluation

While we did not discuss evaluation in simply-typed or polymorphic contexts, ty utilizes evaluation. We will see what this means shor ent evaluation rules and discuss their meanings.

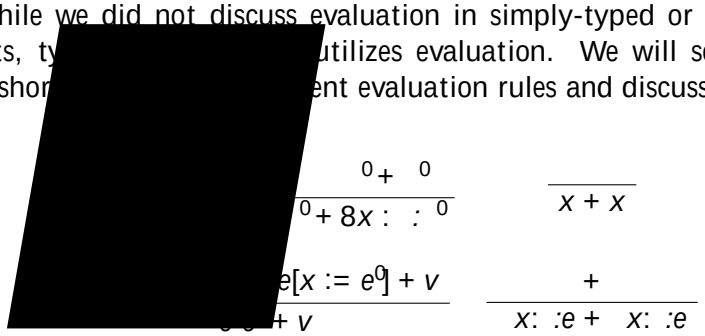

Figure 7: Evaluation Rules for [1]

When we evaluate the *?* term, it simply evaluates to the *?* value.

When evaluating a dependent function space, we evaluate the domain , and the codomain ^{0} to evaluate the whole term.

Evaluating applications, we rst evaluate the operator term to a lambda abstraction value. We then replace all x in the body of the lambda abstraction with e^0, and evaluate this substituted body to vv

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.

$$\frac{}{\Gamma \vdash \star_? : \star_?}\ (\text{Star}) \qquad \frac{\Gamma \vdash A : \star_? \quad \Gamma, x := A \vdash B : \star_?}{\Gamma \vdash (\forall x : A . B) : \star_?}\ (\text{Pi})$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash A : \star_? \quad \Gamma, x := A \vdash e : B}{\Gamma \vdash (\lambda x : A . e) : \forall x : A . B}\ (\text{Lam})$$

$$\frac{\Gamma \vdash e : \forall x : A . B \quad \Gamma \vdash e' : A \quad B[x := e'] \Downarrow C}{\Gamma \vdash e\ e' : C}\ (\text{App})$$

Figure 8: Type Rules for [1]

The Star term type checks as the type $\star_?$

4.1.4 Example Type Derivation in

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

(1)

$$\dfrac{` \ ?:? \qquad ?+? \qquad \dfrac{\dfrac{[x := ?](x) = ?}{[x := ?]\ `\ x : ?} \qquad x + x \qquad \dfrac{[x := ?][y := x](y) = x}{[x := ?][y := x]\ `\ y : x}}{[x := ?]\ `\ \ y{:}x{:}y}}{`\ (\ \ x{:}\,?{:}\ \ y{:}x{:}y) : (8x : ?{:}8y : x{:}x)}$$

This is the polymorphic identity function in . Although it is quite similar to System F, notice that the type parameter x is abstracted with the same lower-case 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:

$$((\ \ x{:}\,?{:}\ \ y{:}x{:}y)\ int)\ 4$$

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:

$$\dfrac{1 \quad `\ (\ \ x{:}\,?{:}\ \ y{:}x{:}y)\ int : (8x : int{:}int) \qquad \dfrac{(4) = int}{`\ 4 : int} \qquad \dfrac{int}{int[x := 4] + int}}{`\ ((\ \ x{:}\,?{:}\ \ y{:}x{:}y)\ int)\ 4 : int}$$

Type Derivation (1)

x:?:1/3:15outputt8Tf 14.246 0 Td [([ )]TJ/F55 10.9091 Tf 3.03 0 Td [(x)]TJ/F15 10.9091 Tf 9.265 0.379

the intention is to gain additional capabilities through the more complex type system. To take full advantage of , 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 *list*(*x*), where has kind *?* and *x* is an integer. In concrete instances, this allows us to construct objects of such types as *int list*(3), or *? 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[ ]	Nil of type
more[*e;*] e^0 e^{00}	Cons of length *e*, type , element e^0, and rest e^{00}
list(*e*)	List of type

$$\frac{\vdash \quad : ? \qquad +}{\vdash \ stop[\ ] : \quad list(0)}\ (\text{Nil})$$

$$\frac{e + v \qquad +}{\vdash e : int \qquad \vdash \quad : ? \qquad \vdash e^0 : \qquad \vdash e^{00} : \quad list(v \quad 1)}\ (\text{Cons})$$
$$\vdash more[e; \ ]\ e^0\ e^{00} : \quad list(v)$$

$$\frac{\vdash \quad : ? \qquad \vdash e : int}{\vdash \quad list(e) : ?}\ (\text{List})$$

Figure 9: Type Rules for List Constructs in

To type check a Nil list, we rst check that has kind *?*, and then evaluate to 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 rst ensure that the provided length term is of type *int*, and evaluate it to *v*to

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?

abstraction obfuscates the underlying semantics of 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 , 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 satis ed, 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 .

```
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

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 as a type system. When we try to componentwise multiply two integer lists of di erent lengths, type checking fails, as we require both lists input to be of length 1.

6 Conclusion

These examples show how e ectively handles the same typed situations as less expressive type theories. Simply-typed and polymorphic terms remain well-typed in , 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 exibility of polymorphic lists and express more about the

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 further to reap its full bene ts.

Though my current implementation introduces some extensions of 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 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 o 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 has de ciencies, 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
  | BoolTyn f Q 9.401482.696 Td [(j)]TJ/F66 8.9664 Tf 10.151 0 Td [(5 0 10.959 l S Q 0 g 0 G q 18(t)]TJ 0 g 0 G 1 g 1 G
```

```
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)  >
```

```
10    interpreter.ml  The apply functions can be used to actually
       put a type checker for (or
      implementation of) a primitive to use.
12
      To extend the basis with new primitives, the list primNames
       must be
14    extended with an appropriate identifier.
    )
16
open Environment;;
18
(
20    This is the master list of names of primitive operators.

22    NB: THESE NAMES ARE LAYED OUT IN A FIXED ORDER!
                                                                    
24  let primOpNames = ["+"; "    "; "  "; "/"; "%"; "      "; "<"; "<="; "=
      "; "<> "; ">"; "    333.G ET q 1359 re f Q 0 g 0 G 1 g 1 G q 1 0 0 1 122.61 549.34 c319.20F6d 0 J 0.398 w 0 0 r
```

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

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

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

```
   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 nn
   "))
        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)
  |
```

```
56        (match rator', rands with
            j BinaryOp f, [a;b]  >
58              (f (fst (normalize env a), fst (normalize env b)),
      env)
            j UnaryOp f, [a]  >
60              (f (fst (normalize env a))), env)
    j Prod x  >
62        (Prod (List.map fst (List.map (normalize env) x)), env)
    j 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  j IntType  >
          (IntType, env)
70  j BoolType  >
          (BoolType, env)
72  j BinaryOp f as x  > (x, env)
    j UnaryOp f as x  > (x, env)
74  j List(typ, len)  >
          (List(fst (normalize env typ), fst (normalize env len)),
      env)
76  j Nil e  > (Nil (fst (normalize env e)), env)
    j Cons(len, typ, el, rest)  >
78        (Cons(fst (normalize env len), fst (normalize env typ),
      fst (normalize env el), fst (normalize env rest)), env)
    j IsNil e  >
80        (match fst (normalize env e) with
            j Nil a as t > (Bool(1), env)
82          j Cons(_, _, _, _)  > (Bool(0), env)
            j _  > raise (Failure "Input cannot normalize nn"))
84  j Head e  >
          (match fst (normalize env e) with
86          j Cons(_, _, e, _)  > (e, env)
            j _  > raise (Failure "Cannot normalize head of anything
       other than non empty list nn"))
88  j Tail e  >
          (match fst (normalize env e) with
90          j Cons(_, _, _, e)  > (e, env)
            j Nil e  > (Nil e, env)
92          j _  > raise (Failure "Cannot normalize tail of anything
       other than a list nn"))
    j _  > raise (Failure "Input cannot normalize nn")
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     | _
```

.

```
    equal' t t' && (equal' (subst [(x, z)] e1) (subst [(x', z)]
    e2))
  in
  equal' (fst (normalize env e1)) (fst (normalize env e2))

let rec infer env = function
  | Var x  ->
      (try lookup_typ x !env
       with Not_found  -> raise (Failure "unknown identifier nn"))
  | Star  -> Star
  | Pi (x, t, e)  ->
      let t' = infer env t in
      let temp = !env in
      extend x t env;
      let e' = infer env e in
      env := temp;
      (match t', e' with
          Star, Star  -> Star
        | _, _  -> raise (Failure "invalid type in dependent
    function spacenn"))
  | Lambda (x, t, e)  ->
      let t' = infer env t in
      let temp = !env in
      extend x t env;
      let e' =
        (try infer env e
         with Failure s  ->
           env := temp;
           raise (Failure ("Input doeise  ((1)]410.959 re f Q 0 g 0 G 1 g 1 G q 1 0 0 1 487.641 341.121 cm |
```

```
        (match (infer env e2), (infer env e3) with
194       j List(t, a), List(t', b)    >
              check_equal env t t';
196           List(t', b)
          j List(_, Int i), List( _, Int j)  > raise (Failure "If
     statement on lists does not type checknn")
198       j _, _  > raise (Failure ("If statement does not type
    checknn" ^ (toString (infer env e2)) ^ "  <>  " ^ (toString (
    infer env e3))))))
  j 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
  j 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 j 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
  j Prod x  > Prod (List.map (infer env) x)
216 j 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  checknn" ^ (
    s))));
224     env := temp;
        t
226 j IntType   >
        Star
228 j BoolType  >
        Star
230 j List(typ, len)   >
        (match infer env typ, infer env len with
232       j Star, IntType   > Star
          j _  > raise (Failure "Input does not type  check as list
    nn"))
234 j 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 checknn"));
            List(typ', len')
        | _, List(t, _)  ->
            List(typ', len')
        | _  -> raise (Failure "Cons does not type checknn"))
  | IsNil e  ->
      (match infer env e with
        | List(_, Int i)  -> BoolType
        | _  -> raise (Failure "Input does not type  checknn"))
  | Head e  ->
      (match infer env e with
        | List(t, _)  ->
            t
        | _  -> raise (Failure "Head does not type checknn"))
  | 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  checknn"))
  | _  ->
      raise (Failure "General input does not type  checknn")


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 expectednn")
   )

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  )
2 (  Lexical analyzer returns one of the tokens:
     the token NUM of integer,
4    operators (PLUS, MINUS, MULTIPLY, DIVIDE, CARET),
     or EOF.  It skips all blanks and tabs, unknown characters.  )
6 f
    open Parser (  Assumes the parser file is "parser.mly".  )
8
```

```
44    | "!="      f NEALT  g
      | ">="      f GE g
46    | ">"       f GT g
      | "true"    f TRUE g
48    | "false"   f FALSE g
      | "let"     f LET g
50    | "in"      f IN g
      | "if"      f IF g
52    | "else"    f ELSE g
      | "then"    f THEN g
54    | "and"     f AND g
      | "or"      f OR g
56    | "not"     f NOT g
      | word+ as string f ID string g
58    | '('       f LPA'
```

```
    | 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
        ) }
```

```
            )
52        )
        )
54    with
          Parsing.Parse_error  >
56          (
              output_string stdout "Input string does not parse
  ...nn";
```