CS345H Final Project Report

Jacob Samuel Van Maynard Geffen
(i.e. Samuel Maynard & Jacob Van Geffen)

December 4, 2015

1 Introduction

L's explosion into popularity has taken the tech world by surprise. A terrible language on almost all accounts, L barely has enough features to qualify as lambda calculus. But this would not be the first time the PL field has been surprised at the rise of a language. Instead of repeating the mistakes of our predecessors and ignoring this new and upcoming language, we hope to bring it closer to being an acceptable language. In this paper we lay out formal semantics for a static type system for L, bringing it out of the unstructured dynamic typing chaos and into clean clear rigid definitions of static types. However we want to do this while impacting current L programmers as little as possible, so we have developed a type inference system for L that will operate without any need for type annotations from programmers.

In addition, we add semantics for arbitrarily polymorphic lists, meaning lists that can contain any type at any part of the list. Thus, it will be possible in new L to have a list of `[5, "duck", lambda x. x+x]`. Finally, we allow for polymorphic list types, as the inability to create a "my_print" function would push programmers away from our language faster than electrons from a negatively polarized magnet.

The meat of this paper is the proof that our new type system obeys both progress and preservation. This means that new L (or as we like to call it JacobSamuelVanMaynardGeffen-L) is type safe, making it one of the very few languages in the world to be so. This is a huge boon for the programming community, as a type safe language means that any program that compiles will never encounter a run time error.

2 Types

In this section, we'll enumerate each of the types we've defined for L and explain their purpose. We'll also discuss our general typing structure, which assumes that all types act like lists.

2.1 List Types

L has the unique feature that both the head (`!`) and tail (`#`) operations are defined over any expressions, not just lists. To account for this, we've decided to implicitly define all types

with a list structure. Every type τ can also be described by some list type `[`τ_1`,` τ_2`]`. For generic list types, this allows us to define polymorphic lists with any arbitrary structure! For atomic types (Constants, Functions, other non-lists), τ_1 `=` τ and τ_2 `= Nil`. As an example, the type `Int` would be equivalent to the type `[Int, Nil]`, `[[Int, Nil], Nil]`, and so on. To formally describe what values the type $[\tau_1, \tau_2]$ encapsulates, we'll define $\gamma([\tau_1, \tau_2]) = \{[v_1, v_2] | v_1 \in \gamma(\tau_1)$ *and* $v_2 \in \gamma(\tau_2)\}$. Here, we're considering `[5, Nil]` to be equivalent to `5`.

2.2 Constant Types

These include `Int`, `String`, and `Nil`. As we discussed in class, an expression cannot have a type that "agrees" with two different Constant types, or a Constant type and a Function type. We'll define this notion of "agreeing" types in a later section.

2.3 Function Types

Any function can be described by $\tau_1 \to \tau_2 \to \ldots \to \tau_k$, where $\tau_1 \ldots \tau_k$ are also types. Notice that $\to$ is **not** associative, so $\tau_1 \to (\tau_2 \to \tau_3) \neq (\tau_1 \to \tau_2) \to \tau_3$.
Note that by our definition of function types, $\gamma(\tau_1 \to \tau_2)$ contains all functions that take elements of $\gamma(\tau_1)$ to elements of $\gamma(\tau_2)$.

2.4 Variable Types

This is our name for polymorphic types. No expression can actually evaluate to a variable type, but variable types **can** be used within functions. For example, a function that prints an expression would evaluate to the type $v \to v$. Notice that in this case, the variable names match, since their concrete types must be equivalent. Some functions may include multiple variable types with different names. A function that took in two values but printed only the first would type to $x \to y \to x$. In practice, functions with variable types are generated when the equivalence class of a variable type (defined by our type rules) only contains other variable types (i.e. no concrete types).
We'll also define how to find γ of some variable function types. $\gamma(x \to x)$ contains all functions that map any concrete value v to elements of $\gamma(\alpha(v))$. $\gamma(x \to y)$ contains all functions that map any concrete value v to any other concrete value u.

2.5 MultiTypes

This is our name for restricted polymorphic types. Like variable types, MultiTypes can accept expressions of multiple types. However, unlike variable types, MultiTypes can only agree with a finite number of types. For example, a function from the MultiType of `String` and `Int` (written `MT(Int|String)` to the same MultiType can accept either a `String` or an `Int`. This function **cannot** accept any other types, such as a list or `Nil`. Multitypes allow us to have polymorphic functions that work for both `Int`s and `String`s, but not for any other type.
For any MultiType $\tau = [\tau_1, \tau_2]$, τ_1 is a MultiType that allows any type that is a head of some

type that τ allows. Likewise, τ_2 is a MultiType that allows any type that is a tail of some type that τ allows.

2.6 Omega Type

There are some L expressions that can't evaluate to a single type (e.g. an `if` that could either evaluate to an `Int` or `Nil`). We can't assume anything about the type of these expressions, but as long as the L program doesn't perform an operator on this expression specific to a single type, the program should still type check successfully. Omega type (Ω) is our solution to this problem. Much like the `Object` type in Java, expressions of type Ω can act whenever **any** other type would also suffice. If a specific type or MultiType is required, any expression of Ω will not type check successfully. We'll show some examples of how Ω is used in a later section.
Note that for $\Omega = [\tau_1, \tau_2]$, $\tau_1 = \Omega$ and $\tau_2 = \Omega$. This is because no information can be assumed about the head or tail of an expression of type Ω.
Another important note to make is that for any type τ, $\gamma(\tau) \subset \gamma(\Omega)$. This is because Ω encapsulates all concrete values.

3 Agreeing Types and Most Specific Union

3.1 Agreement

Agreement is a looser notion of equality that allows us to check if a specific type is allowed by a Variable type or MultiType. If two types agree, then their equivalence classes can be unified without failure. In the typing rules, we'll denote the statement "types τ_1 and τ_2 agree" as $\tau_1 \cup \tau_2$. This is very similar to τ_1 being a "sub-type" of τ_2. However, unlike sub-type relationships, agreement relationships are symmetric. This is necessary because in our type system, it is possible to apply a function of type $(\texttt{Int} \rightarrow \texttt{Int}) \rightarrow \texttt{Int}$ to an expression of type $MT(\texttt{Int}|\texttt{String}) \rightarrow MT(\texttt{Int}|\texttt{String})$.

3.2 Most Specific Union

Our Ω type allows us to type expressions that could evaluate to many different concrete types. An important insight is that an Ω type can only originate from an if statement. We need some operation that determines the tightest generalization of the types of both branches of an if statement. We'll call this operation the "most specific union" of two types (written $msu(\tau_1, \tau_2)$. The most specific union of two types produces a type that is the "least general" (i.e. most specific) generalization of both types. For example, $msu(\texttt{Int}, \texttt{Int})$ produces the type `Int`, but $msu(\texttt{Int}, \texttt{String})$ produces the type Ω. msu is also defined recursively on lists, so $msu([\tau_1, \tau_2], [\tau_3, \tau_4]) = [msu(\tau_1, \tau_3), msu(\tau_2, \tau_4)]$. From this, we can conclude that for types τ_1 and τ_2, $\tau_1 \in msu(\tau_1, \tau_2)$ and $\tau_2 \in msu(\tau_1, \tau_2)$. This fact will be useful in our proof of preservation.

3.3 Type Union Substitution

Since we allow polymophic function types, the function application rules we defined in class won't quite work. Instead, for a function $\tau_1 \to \tau_2$ applied to an expression of τ_3, we need a way of checking that τ_1 and τ_3 agree without actually requiring that the types are equal. We **also** need a way of evaluating the type of τ_2 under the assumption that τ_1 and τ_3 are equal. To do so, we'll define a "type union substitution" operation that evaluates τ_2 under this assumption.

4 Type Rules

Constants and Identifiers

$$\frac{Integer\ i}{\Gamma \vdash i : Int} \quad \frac{String\ s}{\Gamma \vdash s : String} \quad \frac{}{\Gamma \vdash Nil : Nil} \quad \frac{Identifier\ id}{\Gamma \vdash id : \Gamma(id)}$$

Let Statement

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma[x \leftarrow a] \vdash e_2 : \tau_2\ (a\ fresh) \\ a = \tau_1\end{array}}{\Gamma \vdash let\ x\ =\ e_1\ in\ e_2 : \tau_2}$$

Lambda and Application

$$\frac{\Gamma[x \leftarrow a] \vdash e : \tau\ (a\ fresh)}{\Gamma \vdash \lambda x.e : a \to \tau} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \tau_1' \to ... \to \tau_k' \to \tau' \\ \forall i \leq k.\ \Gamma \vdash e_i \vdash \tau_i \\ \forall i \leq k . \tau_i \cup \tau_i'\end{array}}{\Gamma \vdash (e\ e_1\ ...\ e_k) : \tau'[\forall i \leq k . \tau_i \cup \tau_i']}$$

If Statement

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2 \\ \Gamma \vdash p : \tau_p \\ msu(\tau_1, \tau_2) = \tau_3, \tau_p = Int\end{array}}{\Gamma \vdash if\ p\ then\ e_1\ else\ e_2 : \tau_3}$$

Binary Operators (`Int`s only)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2 \\ \tau_1 = Int, \tau_2 = Int\end{array}}{\Gamma \vdash e_1 \odot e_2 : Int}$$

where $\odot$ can be `*`, `-`, `/`, `<`, `<=`, `>`, `>=`, `&`, `|`.

Binary Operators (`Int`s and `String`s)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_1 \\ \tau_1 \ = \ Int \text{ or } String \\ \tau_2 \ = \ Int \text{ or } String \\ a = \tau_1, a = \tau_2 \ (a \ fresh)\end{array}}{\Gamma \vdash e_1 \odot e_2 : a}$$

where $\odot$ can be `+, =, <>`.

List Operators

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1 @ e_2 : [\tau_1, \tau_2]} \quad \frac{\Gamma \vdash e : [\tau_1, \tau_2]}{\Gamma \vdash !e : \tau_1} \quad \frac{\Gamma \vdash e : [\tau_1, \tau_2]}{\Gamma \vdash \#e : \tau_2}$$

Other Unary Operators

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash isNil \ e : Int} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash print \ e : Int}$$

5 Example Programs and their Types

1. The following example demonstrates that the result of a function that evaluates to a *MultiType* can be fed to a type agnostic function.
   ```
   let f = lambda x. if x then 1 else "false" in
   let g = lambda y. print y in
   (g (f 3))
   ```
   evaluates to the type `ConstantType(Int)`, which is the return value of a `print`.

2. This example demonstrates a function which would run correctly, but our type system excludes.
   ```
   let f = if 1 then 3@4 else 5 in
   (#f + 7)
   ```
   The if would always evaluate to a list with two *Ints*, but the type system doesn't know that and rejects this program.

3. This example shows off our *OmegaType*.
   ```
   let f = lambda x. if x then 1@"duck"@4@42 else 3@"duck"@"quack" in
   (f 4)
   ```
   evaluates to `[ConstantType(Int), [ConstantType(String), OmegaType]]`. This not only shows that our system can preserve the *Int* and *String* that both lists have at the head, but that it also does not fail because the lists have different pieces internally. Instead, it goes as far as it can and guarantees nothing beyond that with the *OmegaType*.

6 Modifications to Semantics

6.1 Concatenating `Nil` to a list

We've added a special case rule for concatenation that reads

$$\frac{E \vdash e_1 : e_1'(e_1' isalist) \quad E \vdash e_2 : Nil}{E \vdash e_1 @ e_2 : [e_1', Nil]}$$

This means that `5@Nil` still evaluates to `5`, but `(1@2)@Nil` now evaluates to `[1, 2, Nil]`. If we didn't include this fix, under our typing rules, the tail of `(1@2)@Nil` would incorrectly type to `Nil`.

6.2 Static Scope

We now require that L be statically scoped. This means that in any lambda body, all identifiers used must be previously bound. Identifiers in the body that aren't quantified by the lambda retain the type of their most recent definition.
We found that this addition was necessary for our typing of identifiers to succeed. Since the type of any lambda is evaluated using the lambda body, the type of any identifier used in the body must be previously bound.

7 Proving Soundness

To avoid taking up unnecessary space, we'll reference semantic and type rules without re-typing the rule.

7.1 Showing Progress

Since this is a proof by structural induction, we'll assume that each sub-expression types correctly under our typing rules (and can also be evaluated). Then we'll show that the current expression can also be evaluated if it can be successfully typed.

1. **Constants and Identifiers**
   All constants successfully type, and all constants can be evaluated, so progress is held. Identifiers type only if the identifier is bound by Γ. Since any identifier bound by Γ must also be bound by E (see rules for let, lambdas, and application), id's can also only be evaluated if they are bound.

2. **Let Statements**
   For a statement `let x = a in b` that types successfully, we can assume that `a` types to type τ_1 and that `b` types to type τ_2 when `x` types to τ_1. Since `a` types, we know that `a` evaluates to some `e` of type τ_1. Remember that the above statement can be evaluated

if and only if `b[x←e]` can be evaluated. This condition is satisfied if we know that `b` types successfully when `x` has the same type as `e`. Our assumptions guarantee this, so the statement can be evaluated.

3. **If Statements**
   For a statement `if a then b else c`, we know that `a` must type to Int, `b` must type to τ_1, and `c` must type to τ_2. The if statement can always be evaluated as long as `a` evaluates to an Int and both `b` and `c` evaluate successfully. By our assumptions, we know both of these statements are true. Thus, we can evaluate this if statement.

4. **Lambdas**
   Any lambda expression in L will evaluate successfully as long as the body doesn't use any unbound identifiers (since L is now statically scoped). Lambdas will also not type successfully if their body uses an unbound identifier. Thus, any lambda statement that successfully types can also be evaluated.

5. **Application**
   For a statement `(e` $\mathtt{e}_1$ `...` $\mathtt{e}_k$`)` that types successfully, we can assume that `e` types to $\tau_1' \rightarrow ... \rightarrow \tau_k' \rightarrow \tau$ and that each $\mathtt{e}_i$ types to τ_i. We can also assume that for each i, $\tau_i = \tau_i'$, since `(e` $\mathtt{e}_1$ `...` $\mathtt{e}_k$`)` only types if we can make this assumption (see "Agreement"). Since each $\mathtt{e}_i$ types, we know that $\mathtt{e}_i$ evaluates to some $\mathtt{e}_i$`'`. Under our operational semantics, this expression can evaluate as long as `e[`$\forall i.\mathtt{x}_i \leftarrow \mathtt{e}_i$`']` can be evaluated. Since `(e` $\mathtt{e}_1$ `...` $\mathtt{e}_k$`)` types under the assumption that each τ_i and τ_i' agree, we know this statement is true. So `(e` $\mathtt{e}_1$ `...` $\mathtt{e}_k$`)` can be evaluated.

6. **Binary Operations over Integers**
   Under our type rules, the expression `a`$\odot$`b` types successfully as long as both `a` and `b` type to Int (the possible replacements for $\odot$ are discussed in the type rules above). This means that both `a` and `b` can be evaluated, and we can perform the $\odot$ operation to both. Thus, under our evaluation semantics, `a`$\odot$`b` can also be evaluated.

7. **Binary Operations over Integers and Strings**
   Under our type rules, the expression `a`$\odot$`b` types successfully as long as both `a` and `b` type to Int or $String$ (and their types are equal). The possible replacements for $\odot$ are discussed in the type rules above. This means that both `a` and `b` can be evaluated, and we can perform the $\odot$ operation to both. Thus, under our evaluation semantics, `a`$\odot$`b` can also be evaluated.

8. **Concatenation**
   Under our type rules, the expression `a@b` types successfully as long as both `a` and `b` type successfully. This means that both `a` and `b` can be evaluated. Thus, under our evaluation semantics, `a@b` can also be evaluated.

9. **Head and Tail**
   The expressions `!e` and `#e` both type successfully as long as `e` types successfully. Likewise, `!e` and `#e` can both be evaluated as long as `e` can be evaluated. Thus, since `e` types, it can be evaluated, so both expressions can also be evaluated.

10. **isNil and print**
    The expressions `isNil e` and `print e` will both evaluate as long as `e` also successfully evaluates. By our type rules for `isNil` and `print` type `e` to type τ, so `e` can be evaluated in both cases. Thus, both `isNil e` and `print e` can be evaluated.

7.2 Showing Preservation

Since this is a proof by structural induction, we'll assume that each sub-expression hold preservation. Then we'll show that if the current expression evaluates to v and types to τ, then $v \in \gamma(\tau)$.

1. **Constants and Identifiers**
   Our operational semantics specify that any constant evaluates to itself. Each constant type includes all constants of that category. Thus, for any constant v and it's corresponding type τ, $v \in \gamma(\tau)$.
   To show preservation for identifiers, we'll have to show agreement. For all rules that don't change E or Γ, we'll inductively assume that E and Γ agree. As we discussed in class, for each of the remaining rules, we'll assume preservation holds.
   For any let statement `let x = a in b`, assume that `a` evaluates to v and types to τ_a. We change both E and Γ so that $E(x) = v$ and $\Gamma(x) = \tau_a$. Due to preservation, $v \in \tau_a$, so $E(x) \in \gamma(\Gamma(x))$.
   For any application statement `(e1 e2)`, we know that $\texttt{e}_1$ evaluates to `lambda x. e` and types to $\tau_2' \to \tau'$, and that $\texttt{e}_2$ evaluates to v and types to τ_2. We change $E(x) = v$ and $\Gamma(x) = \tau_2'$. Since we have preservation, we know that $v \in \gamma(\tau_2)$. By our assumption that τ_2 and τ_2' agree, this implies that $v \in \gamma(\tau_2')$. Thus, $E(x) \in \gamma(\Gamma(x))$.
   These are the only expressions that change E and Γ, so we have shown agreement.

2. **Let Statements**
   Consider an expression `let x = a in b` that evaluates to v and types to τ. We can also assume that `a` evaluates to v_a and types to τ_a, where $v_a \in \gamma(\tau_a)$. This means that `b[x`$\to v_a$`]` evaluates to v and types to τ. Since our sub-expressions satisfy preservation (by our inductive hypothesis), this means that $v \in \gamma(\tau)$.

3. **If Statements**
   Consider an expression `if a then b else c`. We'll assume the expression types to type τ and evaluates to v, so `b` and `c` must type to τ_b and τ_c respectively. By our inductive hypothesis, if `b` and `c` evaluate to v_b and v_c, then $v_b \in \gamma(\tau_b)$ and $v_c \in \gamma(\tau_c)$. Thus, by our definition of msu, and since $\tau = msu(\tau_b, \tau_c)$, $v_b \in \gamma(\tau)$ and $v_c \in \gamma(\tau)$. `if a then b else c` evaluates to v, which is equal to either v_b or v_c, so $v \in \gamma(\tau)$.

4. **Lambdas**
   Any expression `lambda x. e` can be evaluated, no matter the type (since it evaluates to itself). Let's say that this expression types to $\tau_1 \to \tau_2$. We then know that `e` types to τ_2 when any occurrence of `x` in `e` types to τ_1. By definition, this means that `lambda x. e` is a function that takes elements of $\gamma(\tau_1)$ to elements of $\gamma(\tau_2)$. As we discussed in our section on function types, this means that (`lambda x. e`)$\in \gamma(\tau_1 \to \tau_2)$.

5. **Application**
   Consider an expression `(e e1 ... ek)` that evaluates to v and types to τ. We can assume that each `ei` evaluates to v_i and types to τ_i, and also that $v_i \in \gamma(\tau_i)$. Since `e` types to $\tau_1 \to ... \to \tau_k \to \tau$, we know that `e` evaluates to a function that maps tuples in $\gamma(\tau_1) \times ... \times \gamma(\tau_k)$ into elements of $\gamma(\tau)$. Since `(e1, ..., ek)` is in $\gamma(\tau_1) \times ... \times \gamma(\tau_k)$ by our inductive hypothesis, v must be in $\gamma(\tau)$.

6. **Binary Operations over Integers**
   Consider an expression `a`$\odot$`b` that evaluates to v and types to Int. We can also assume that `a` and `b` evaluate to integers (elements of $\gamma(Int)$). Thus, the operation `a`$\odot$`b` mathematically produces an integer i, and $i \in \gamma(Int)$, so this operation preserves type.

7. **Binary Operations over Integers and Strings**
   Consider an expression `a`$\odot$`b` that evaluates to v and types to Int or to $String$. We can also assume that `a` and `b` evaluate to integers or strings (elements of $\gamma(Int)$ or $\gamma(String)$). Thus, the operation `a`$\odot$`b` produces either an integer i or a string s. Either way, $i \in \gamma(Int)$ and $s \in \gamma(String)$, so this operation preserves type.

8. **Concatenation**
   Consider an expression `a@b` that evaluates to $[v_h, v_t]$ and types to $[\tau_1, \tau_2]$. We know that `a` evaluates to v_h and types to τ_1, while `b` evaluates to v_t and types to τ_2. Thus, $v_h \in \gamma(\tau_1)$ and $v_t \in \gamma(\tau_2)$. Remember that $\gamma([\tau_1, \tau_2]) = \{[v_1, v_2] | v_1 \in \gamma(\tau_1)$ *and* $v_2 \in \gamma(\tau_2)\}$, so $[v_h, v_t] \in \gamma([\tau_1, \tau_2])$.

9. **Head and Tail**
   For both expressions `!e` and `#e`, we'll assume that `e` evaluates to $[v_h, v_t]$ and types to $[\tau_1, \tau_2]$. We'll also say that `!e` evaluates to v_h and `#e` evaluates to v_t (by our operational semantics). In our section on list types, we noted that $\gamma([\tau_1, \tau_2]) = \{[v_1, v_2] | v_1 \in \gamma(\tau_1)$ *and* $v_2 \in \gamma(\tau_2)\}$. By our inductive hypothesis, we can assume that $[v_h, v_t] \in \gamma(\tau_1)$. Thus, $v_h \in \gamma(\tau_1)$ and $v_t \in \gamma(\tau_2)$.

10. **isNil and print**
    Both expressions `isNil e` and `print e` evaluate to integers and type to Int. As we discussed in the section on Constants, for any integer i, $i \in \gamma(Int)$.

8 Modifications to Test Cases

Actual modifications can be found in the test folder.

8.1 Test 4

Under our type rules, this program does not type successfully. This is because `g` is undefined in the function `f`. Under our type rules, when `lambda x. ... (g x)` is typed, since $\Gamma(g)$ is undefined, the program fails to type. We decided this was an acceptable change because under the proposed rules in the slides, this program fails to type as well.

8.2 Test 5

Due to our changes in semantics, we've modified the base case of `read_list` to return an `Int`. This way, we can infer that the return value of `read_list` is of type `[Int, Ω]`. We've also changed `l+n` in `add` to `!l+n`, because at this point, all the type system knows about `l` is that `l`'s head is an `Int`.

9 Other Shortcomings and Possible Future Work

9.1 Unbound Identifiers in Lambdas

As we mentioned in the above section, though unbound identifiers would originally evaluate perfectly fine, they will not type check properly. We believe that our change to static scoping is well founded, but we do lose some expressivity. Programmers can no longer write functions with unbound identifiers in the hopes that the id's will be defined before the function is used. However, static scoping does allow for cleaner and easier-to-read code, so we think this change is acceptable.

9.2 Casting

One feature we'd like to add if given more time is type casting. This would allow users to cast expressions with general types (like Ω) to concrete atomic and list types. This would allow programmers to use expressions with type Ω. However, doing so would void the soundness of our type system. We'd still be able to prevent impossible casts (like casting `"duck"` to an `Int`), but other casts might fail at run-time instead. Still, casting would allow for even more expressivity in our language.

10 Conclusion

"I am the Alpha and the Omega, the first and the last, the beginning and the end" - Revelation 22:13. We were given the α type to represent any possible type, but this did not suit our needs. We are not accepting gods, we are wrathful and jealous gods. We needed an ending type. A type which would accept no other type. We needed the Ω. The last. The end. With Ω we can allow for the creation of functions statements that return a *String* or an Int or a Nil, something programmers commonly need, but also make sure that no invalid operations are used on this 'unknown' type. Truly, Ω is our crowning achievement.

Oh, we also did completely polymorphic list types I guess. Those are cool. Lists are no longer restricted to just $ConstantTypes$, they can be lists of $FunctionTypes$, $ListTypes$, $ConstantTypes$, even $OmegaTypes$ interspersed randomly as desired. This is extremely useful because it allowed for more complex data structures, like maps or trees. Random access polymorphism is accomplished by removing the difference between atoms and lists. By making every atom behave like a list we can avoid the need for two rules for processing head and tail operations, the primary difficulty in making polymorphic lists.

Our third, and perhaps most underwhelming when compared with the power of Ω, accomplishment is polymorphic functions. Polymorphic functions fall out of the $OmegaType$ and $MultiType$ definitions. If a function can evaluate to an Int or a $String$ we say the return type is a $MultiType$ of Int and $String$. Note that this can be used arbitrarily, so that in theory a function could return a $MulitType$ of Int and $Int \rightarrow Int$, it's just that this particular $MultiType$ is the equivalent of Ω (since no operation besides a type agnostic one such as `print` can be applied to it).

We hope that these additions to the L language will increase the already surging popularity of the language, and that programmers will not be displeased with our copyright on this new version of L: JacobSamuelVanMaynardGeffen©.

11 Running our Tests

`make test`
The expected output is contained in a comment in the top of every file.