L - a fantastic lambda-calculus-based functional programming language

Patricio Lankenau and Robert Perce

December 4, 2015

Abstract

In this paper, we introduce L, a fantastic functional programming language based on the lambda calculus. L is statically typed with full Hindley-Milner based type inference, which requires no type annotations. Furthermore, L provides many features to the developer such as beautiful syntax, beautifully designed operational semantics, full stream operators on lists, and much more! In this paper we describe our contributions as well as the design decisions behind them. We then go on to formally prove that our type system is sound and powerful enough to prevent all run-time errors. Finally, we give some example of our beautiful language performing type inference on a wide variety of programs, as well as examples of where our type inference engine prevent run-time errors that competing languages would have let slip.

1 Syntactic Sugar

In order to make L more appealing to all future L programmers—and because we have Haskell experience—we decided to create a shorthand notation defining lambda functions. Instead of writing out, e.g. `lambda x.x`, a programmer may now use `\x.x`.

Furthermore, in order to remove confusion about the unary operators for head and tail we created a more readable thus more useful way of invoking them using named functions: `head` and `!` both lex to `TOKEN_HD`, and `tail` and `#` are `TOKEN_TL`.

Finally, to make code more readable we added the ability to initialize lists as a literal construct—see below.

2 List Semantics

- Allow `[e1, e2, e3]` to create a list.
- `Nil` can be written as `[]`.
- Distinguish between a singleton list and the element inside, such that `![a]` is meaningful.
- The `cons` operator now takes a `List[T]` on the right and a `T` on the left, and prepend the element to the list, e.g. `a @ [b, c] = [a, b, c]`. This is exactly the way Haskell handles its `cons` operator, `:`.

2.1 Static list initialization

Like most real programming languages, we added support for list initialization using a literal syntax. A list may be created via a comma-separated list of expressions wrapped in square brackets, e.g. `[0, 1, 2, 3]`—much cleaner and more pleasant than the `cons` approach, which would have in this example appeared as `0 @ 1 @ 2 @ 3 @ Nil`.

2.2 More efficient storage (using vectors)

Rather than a `AstList` begin a glorified tuple, each `AstList` is now backed by a `std::vector`, which means prepending and iterating is quite fast.

2.3 Rewriting of Cons

This required changing the `CONS` section of `AstBinOp` to prepend the left argument onto the right argument. Typing let us specify that cons is type `T` $\rightarrow$ `[T]`, where `T` is an `ArbitraryType` (see below for details on that).

2.3.1 New operational semantics

$$\frac{E \vdash e_1 : v_1 \quad E \vdash e_2 : [L]}{E \vdash e_1 \texttt{ @ } e_2 : [v_1, L]} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{List}[\tau]}{\Gamma \vdash e_1 \texttt{ @ } e_2 : \texttt{List}[\tau]}$$

$$\frac{\begin{array}{c} E \vdash e_1 : v_1 \\ E \vdash e_2 : v_2 \\ \vdots \\ E \vdash e_n : v_n \end{array}}{E \vdash [e_1, e_2, \ldots, e_n] : [v_1, v_2, \ldots, v_n]} \qquad \frac{\begin{array}{c} \Gamma \vdash e_1 : \tau \\ \Gamma \vdash e_2 : \tau \\ \vdots \\ \Gamma \vdash e_n : \tau \end{array}}{E \vdash [e_1, e_2, \ldots, e_n] : \texttt{List}[\tau]}$$

2.3.2 Problems with defining both append and prepend

Initially, we wanted `@` to perform both element *a*ppending and *pre*pending... before we discovered that would be somewhere between "excruciatingly difficult" and "actually impossible" to type. So we settled on prepend, in part because It's Like Haskell So It Can't Be All Wrong, and in part because `1 @ 2 @ 3 @ Nil` looks a bit nicer than `Nil @ 1 @ 2 @ 3`.

3 Stream Support

Our final addition to the language is powerful support for stream operators much like other established languages like Java, python, and Haskell have.

3.1 Map

We added the ability to map over lists with arbitrary functions. Map therefore takes a function $\tau_1 \to \tau_2$ and a list of type τ_1 and returns a list of type τ_2. This is an extremely useful construct that can be used to process and modify lists.

```
map \x.x+1 [1,2,3]
```

```
final solved type:      List[ConstantType(Int)]
[2, 3, 4]
```

3.2 Filter

We also added the ability to filter lists by application a predicate function to each element. This returns a new list containing the elements which yielded a truthy predicate result. This will take a predicate function $\tau_1 \to Int$ and a list of τ_1 and return a new list of τ_1.

```
filter \x.x>10 [1,8,99,4,55,10,100]
```

```
final solved type:      List[ConstantType(Int)]
[99, 55, 100]
```

3.3 Reduce

Finally, we added the ability to reduce a list. This takes an accumulator (initial value) of type τ_1, a function which takes two parameters $\tau_1 \to \tau_1 \to \tau_1$, and a list of type τ_1. It will then being to apply the function to the accumulator and each element within the list (starting with the accumulator and the first, then that result and the second, etc...), and finally return the final accumulator.

```
reduce "" \x,y.x+y ["r", "o", "b", "e", "r", "t"]
```

```
final solved type:      ConstantType(String)
"robert"
```

4 Type System

4.1 Types

- `ConstantType(Int)`

  A constant type that represents all positive and negative integral values between $-2^{15}+1$ and $2^{15}-1$. In this document, unless we specify "in code output", we use `ConstantType(Int)` and `Int` to represent the same thing.

- `ConstantType(String)`

  A constant type that represents arbitrary length strings comprised of characters, exactly as one would probably expect. In this document,

unless we specify "in code output", we use `ConstantType(String)` and `String` to represent the same thing.

- `VariableType`

  Represents a variable in the program. These are annotated with the name of the identifier. In output, may be seen with one or more appended apostrophes, `'`. This is done reminiscent of alpha reduction to prevent name conflicts.

- `ArbitraryType`

  Represents any type. Essentially used to create fresh types—a type variable that doesn't rely on abusing the name of a `VariableType`.

- `FunctionType`

  Represents any function (either lambda or operator) that takes some type and maps to some type (potentially another function type). For example, a function which takes an `Int` and returns an `Int` would have function type `Int` $\rightarrow$ `Int` whereas a function that takes two `Int`s and returns a `String` would have function type `Int` $\rightarrow$ `Int` $\rightarrow$ `String`, seen in code output as `FunctionType(`*`name`*`, ConstantType(Int), FunctionType(`*`name`*`, ConstantType(Int), ConstantType(String)`, since there's not really any such thing as a function of multiple variables—it's just syntactic sugar for currying.

  Internally, the sequences of types is stored in order in a `std::vector`.

- `ListType`

  Represents a list. Each list type keeps track of the subtype—the type of the contained elements. This means that we allow for recursive lists. For example, a list of lists of `Int`s is entirely valid, and has type `List[List[Int]]`.

- `TypeClass`

  Type classes represent a set of types. These are used to correctly type operators which are polymorphic over a subset of all types. We support the following two typeclasses:

  - `TypeClass::Plussable`

Elements that can be added, such as integers (under integer addition) and strings (under string concatenation).

- `TypeClass::Equatable`

  Elements that can be equated using the equals operator. Currently, this is again only integers and strings, but it's not overly difficult to implement equality of other types as well.

- WeakType

  The WeakType is a type wrapper that is used when building a constraint from a function application, like `(\x.x 6)`. Since functions—like `\x.x` in the example—may be polymorphic, it wouldn't do to simply declare that the type of `\x.x` must be `Int` $\rightarrow \tau$. A program might use the same polymorphic function on multiple types, as in `let id = \x.x in let _ = (id "foo") in (id 3)`, and here the constraints that `id` was `Int` $\rightarrow \tau$ and `String` $\rightarrow \tau$ conflict, even though usage was perfectly valid.

  To avoid this problem, we instead constrain that a function is type `WeakType(`*`input`*`)` $\rightarrow \tau$. Now, if two `WeakType`s disagree—like in the above example, where we attempt to unify `WeakType(Int)` and `WeakType(String)`—they both give up, and unify to a new `ArbitraryType`. If a `WeakType` attempts to unify with a non-weak type that can unify with the wrapped type, the parent becomes that non-weak unification. If a `WeakType` attempts to unify with a non-weak type that *cannot* unify with the wrapped type, the unification fails, and a type error is reported.

  Now we can guarantee type *safety*, but the final reported type in cases like the above, where the ultimate result of the program is determined by a polymorphic application in a program where the function is used polymorphically, is now an `ArbitraryType`, even though running the function clearly produces a concrete type. To fix *this* problem, we note that every time the body of a `let` binding is another `let` binding, we must not yet be at the terminus of the program. Once that end *is* reached, we mark any generated `WeakTypes` as `is_final`. A `WeakType` that is marked `is_final` behaves identically to a non-`final` `WeakType`, except for one case: when a `final` `WeakType` disagrees with a non-`final` `WeakType`, the `is_final` one wins out, and they both unify to that. If

two `is_final` `WeakType`s disagree, they still both unify to an arbitrary type (which, in a valid type-checking program, can't happen).

Now, cases ending in a `WeakType` produce the correct value, and at final report time if the ultimate final type of the program, after all `union`s and `find`s complete, is a `WeakType`, it is unrolled. It was unsure of itself, but it made it all the way to the end, so it must be right (this is also what happens when a function is used *non*-polymorphically, so all the generated `WeakType`s for that function agree with each other).

4.2 Type Rules

4.2.1 Base Cases

$$\frac{i = \text{int}}{\Gamma \vdash i : \texttt{Int}} \qquad \frac{s = \text{string}}{\Gamma \vdash s : \texttt{String}} \qquad \frac{id = \text{identifier} \quad \Gamma(id) = \tau}{\Gamma \vdash id : \tau} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma \vdash e_2 : \tau \\ \vdots \\ \Gamma \vdash e_n : \tau\end{array}}{\Gamma \vdash [e_1, e_2, \ldots, e_n] : \texttt{List}[\tau]}$$

4.2.2 Binary Operators

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : Int \\ \Gamma \vdash S_2 : Int \\ \odot \text{ is either -,*,/,\&,|,<,>}\end{array}}{\Gamma \vdash S_1 \odot S_2 : Int} \qquad \frac{\begin{array}{c}\Gamma \vdash S_1 : \tau \\ \Gamma \vdash S_2 : \texttt{List}[\tau]\end{array}}{\Gamma \vdash S_1 @ S_2 : \texttt{List}[\tau]}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau \\ \Gamma \vdash S_2 : \tau \\ \tau \in \texttt{Plussable}\end{array}}{\Gamma \vdash S_1 + S_2 : \tau} \qquad \frac{\begin{array}{c}\Gamma \vdash S_1 : \tau \\ \Gamma \vdash S_2 : \tau \\ \tau \in \texttt{Equatable} \\ \odot \text{ is either } =, <>\end{array}}{\Gamma \vdash S_1 \odot S_2 : Int}$$

4.2.3 Unary Operators

$$\frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{isNil } S_1 : Int} \qquad \frac{\Gamma \vdash S_1 : \tau}{\Gamma \vdash \text{print } S_1 : Int} \qquad \frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{head } S_1 : \tau} \qquad \frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{tail } S_1 : List(\tau)}$$

4.2.4 Branch

$$\frac{\Gamma \vdash p : Int \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : \tau}$$

4.2.5 Application

$$\frac{\Gamma \vdash S_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash S_2 : \tau_1}{\Gamma \vdash (S_1 S_2) : \tau_2}$$

4.2.6 Lambda

$$\frac{\text{will be of type } \tau_1 \quad \Gamma[x \leftarrow \tau_1]S_1 : \tau_2}{\Gamma \vdash \lambda x.S_1 : \tau_1 \to \tau_2}$$

4.2.7 Let

$$\frac{\Gamma \vdash S_1 : \tau \quad \Gamma[id \leftarrow \tau]S_2 : \tau_2}{\Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau_2}$$

4.3 Proof of Preservation and Progress

Now that we have our types and type rules defined, we want to prove the preservation of our type system. In other words, we want to prove that the our type system is sound and that the soundness is preserved under transition rules. This is done by inductively showing that for well-typed program e if $E \vdash e : v$ and $\Gamma \vdash e : \tau$ then $v \in \gamma(\tau)$ where γ is the concretization function.

Furthermore, we also want to prove that our type system obeys progress. This is essentially proving that our type system is powerful enough to prevent all run-time errors. This can be done by proving that for each of transitions, if the type system checks then the operational semantics will not get stuck.

We begin by arguing preservation and progress for all four base cases: ints, strings, identifiers, and lists.

$$\frac{i = \text{int}}{\Gamma \vdash i : \text{Int}} \qquad \frac{s = \text{string}}{\Gamma \vdash s : \text{String}}$$

For ints and strings, we simply have the integer value of integers and string value of strings. We must show that these have the type of int and string, which is vacuously true given the hypothesis of the operational semantics and type rules.

The operational semantics will clearly never get stuck if the types are the correct ones.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma \vdash e_2 : \tau \\ \vdots \\ \Gamma \vdash e_n : \tau\end{array}}{\Gamma \vdash [e_1, e_2, \ldots, e_n] : List(\tau)}$$

The same applies for lists, because of the hypothesis we know that we have a list of some number of elements, and for the type we know that the type of each element is the same and therefore the final type is a list of that type.

¡¡¡¡¡¡¡ HEAD Furthermore, if the type is correctly a list the operational semantics will also not get stuck. ======= Furthermore, if the type is correctly a list the operational semantics will also not get stuck. ¿¿¿¿¿¿¿ a6a2140d810f255488803f03dcc022354b941eb7

For identifiers, we will need to prove agreement, so we defer this to a later part of the proof.

Now that we have proved the base cases, we can prove the inductive rules by assuming that preservation and progress holds for all subexpressions and proving they hold for the current expression.

For the binary operators, we begin by proving the integer arithmetic operators.

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : Int \\ \Gamma \vdash S_2 : Int \\ \odot \text{ is either -,*,/,\&,|,<,>}\end{array}}{\Gamma \vdash S_1 \odot S_2 : Int}$$

By the inductive hypothesis we know that the type of the two parameters must be integers (by the hypothesis defined in the operational semantics and in the typing rules). Then, we know that any integer minus another integer will always yield in an integer. Furthermore, the same will hold true for integer multiplication, or integer division. Integer comparisons and boolean operators will also always yield a boolean, which is represented by an integral value. Therefore if the expression correctly types to an Int then we know that the final evaluation of the expression will have to be an integer.

Furthermore, we know that if the type could be computed then both parameters had to be integers. Because the only hypothesis for the operational semantics are that the parameters are integers, they will clearly not get stuck.

$$\frac{\Gamma \vdash S_1 : \tau \quad \Gamma \vdash S_2 : \tau \quad \tau \in \texttt{Plussable}}{\Gamma \vdash S_1 + S_2 : \tau} \qquad \frac{\Gamma \vdash S_1 : \tau \quad \Gamma \vdash S_2 : \tau \quad \tau \in \texttt{Equatable} \quad \odot \text{ is either } =, <>}{\Gamma \vdash S_1 \odot S_2 : Int}$$

The semantics of the plus operator and equality operators have to be separated from the rest of the arithmetic operators because they are defined over a class of types. In our implementation we added typeclass support for things that can be added and things that can be compared, although in L these two type classes are equivalent since they both contain integers and strings.

By the inductive hypothesis we know that the type of both parameters must be equal, and must be within the typeclass. We know this because the operational semantics have hypotheses which define the semantics only for integers and strings. Furthermore we know string concatenation will always yield a string. Similarly integer addition will always yield an integer. For the equatable classes, the operational semantics will always yield a boolean value encoded as as integer as long as the inputs are either ints or strings, which is equivalent to the type checking rules. Therefore the evaluation of an expression will always agree with the type computed and more importantly the expression will always succeed if the type can be calculated since all hypotheses for the operational semantics are met.

$$\frac{\Gamma \vdash S_1 : \tau \quad \Gamma \vdash S_2 : \texttt{List}[\tau]}{\Gamma \vdash S_1 @ S_2 : \texttt{List}[\tau]}$$

The final binary operator is the cons operator. By the inductive hypothesis we know that the type of the first parameter will always be some type τ and the value of the second will be a list, since that it is how it is defined in the operational semantics. If the program type checks, that is the type of the first parameter is the subtype of the list type of the second, then we know that the computed type is valid for the evaluated type.

In similar fashion to previous progress arguments, we know that if the type was computed, that the type of the first parameter is the same as the inner type of the second parameter (which is guaranteed to be a list). Therefore the hypotheses for the operational semantics are met and it will not get stuck.

Next up are the unary operators.

$$\frac{\Gamma \vdash S_1 : \tau}{\Gamma \vdash \texttt{print}\ S_1 : Int}$$

For the print function, it doesn't matter what the type of the expression is as long as it has a type. The operational semantics define print to always return 0, which means that the final type of Int is always correct and because it has no hypothesis, it trivially will never stuck.

$$\frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{isNil}\ S_1 : Int}$$

For the isNil function, the operational semantics check if the parameter is a nil value and return a 1 if it is, otherwise a 0. The type rules will always return a type Int, which agrees. Similarly, the type of list of any type agrees with the parameter if it can be checked against nil.

Because the type rules require the parameter to be a list of some type, and the operational semantics only the require the parameter to be a list, it is guaranteed that if the expression typed, it will not get stuck.

$$\frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{head}\ S_1 : \tau} \quad \frac{\Gamma \vdash S_1 : List(\tau)}{\Gamma \vdash \text{tail}\ S_1 : List(\tau)}$$

Head and tail are defined in the operational semantics to return the first element, or the remainder of the list respectively. The head will always have the same type as all elements inside the list and the tail will always have the type of same type as the original list. Therefore the operational semantics and the typing rules are sound.

If the type of the expression was successfully computed, then we know that the type of the parameter is a list. This is the only hypothesis for the operational semantics, so they will not get stuck.

$$\frac{\begin{array}{c}\Gamma \vdash p : Int \\ \Gamma \vdash e_1 : \tau \\ \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : \tau}$$

The operational semantics of branch define the predicate to be an integer, then the evaluation will be either the first branch if the integer was greater than 0 or the second if it was 0. This is consistent with the typing rules which require the predicate to be of type Int, and return the type of a branch. The typing rules; however, restrict the branches to have the same type in order to perform inference, this is correct but limits the number of programs that can type check. However it gives us the guarantee that if the program type checks then the final type agrees with the final evaluation.

The only requirement for the operational semantics is that the predicate evaluates to an integer. The typing rules require that, so if the program types, the semantics will succeed and progress is preserved.

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash S_2 : \tau_1\end{array}}{\Gamma \vdash (S_1 S_2) : \tau_2}$$

The operational semantics for application requires the first expression to evaluated to a lambda, then it applies the second expression to the body and returns the evaluated result. Lambdas have function types, which is consistent with the type rules. Furthermore, the type rules require that the type of the second parameter match the first type of the function type, then the final type is the second type of the function type. This means that if the type system can calculate a final type, then the application can be performed and the final evaluated value will be in the discretization of the final type.

The requirements for the operational semantics are that the type of the first parameter is a function and that the second parameter can be correctly applied to it. The typing rules require the first parameter to be a function type from τ_1 to τ_2 and the second parameter to be of type τ_1. Therefore if the types rule pass, the operational semantics too will pass.

$$\frac{\begin{array}{c}\text{will be of type } \tau_1 \\ \Gamma[x \leftarrow \tau_1]S_1 : \tau_2\end{array}}{\Gamma \vdash \lambda x.S_1 : \tau_1 \rightarrow \tau_2}$$

Lambdas do not undergo any evaluation so the operational semantics just return the lambda. However, the type semantics have the crucial component of taking the type of the formal and binding it to the identifier under the typing environment. It then uses this new type environment and calculates the type of the body of the lambda. The final return type is therefore a function type from the type of the formal to the type of the body which trivially proves that it is sound with the operational semantics.

Because the operational semantics have no requirements, they will trivially never get stuck.

To prove the final two transitions, we need to prove that the environment and type environment agree by showing that for any identifier x we have that $\Gamma(x) = \alpha(E(x))$ where α is the abstraction function. If we can establish this we have $\Gamma \sim E$ and have proven agreement.

Since the only rules that change the type environment and let and lambda, we can prove that that the two environments agree inductively by building up from the base case.

The base case is when both environments are empty, in which case they trivially agree.

$$\frac{\begin{array}{c}\text{will be of type } \tau_1 \\ \Gamma[x \leftarrow \tau_1]S_1 : \tau_2\end{array}}{\Gamma \vdash \lambda x.S_1 : \tau_1 \rightarrow \tau_2}$$

For the inductive case we start by proving the lambda transition. We can assume preservation, so we know that the type of the argument is some τ. The type rules will place this τ in the type environment bound to the variable. Since this argument is dynamically bound to the variable in the environment and the type in the type environment is τ and they are the same we have shown that $\Gamma \sim E$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \tau \\ \Gamma[id \leftarrow \tau]S_2 : \tau_2\end{array}}{\Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau_2}$$

For the let case, we can make a very similar argument. Assuming preservation, we know that the abstraction of the evaluated value is τ. The typing

rules will bind this τ to the identifier in the type environment. Similarly the operational semantics will bind the evaluated value to the environment. Because we have shown that the type in the type environment agrees with the abstraction of the expression, we have that $\Gamma \sim E$.

Therefore we have shown that agreement holds under all relevant transitions, assuming that agreement held before. Since we have also shown that agreement holds in the base case, we have inductively proven it will always hold under valid transitions.

Now that we have proven agreement, we can use it to finish proving preservation for the last two transitions.

$$\frac{\Gamma \vdash S_1 : \tau \quad \Gamma[id \leftarrow \tau]S_2 : \tau_2}{\Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau_2} \qquad \frac{id = \text{identifier} \quad \Gamma(id) = \tau}{\Gamma \vdash id : \tau}$$

For the identifier base case, we can assume agreement so we know know that the type of the identifier will be sound since know $\Gamma \sim E$.

Furthermore, by assuming agreement we know that if the typing environment has a binding for an identifier then the environment too will have a binding, therefore the operational semantics will not get stuck.

For the let case, we can assume agreement so we know that the abstraction of the final evaluated expression will be consistent since we know the type of the value bound to the identifier agrees. This is exactly what we wanted to prove.

To prove progress for let, we know that the if the typing rules apply then the binding for the identifier also exists in the identifier, therefore the operational semantics will not get stuck.

We have now shown that we can prove preservation and agreement for all of the typing rules. Therefore we can conclude that the abstract value we compute will always be a correct approximation for the concrete value of the program. In other words the type computed will always have the final evaluated expression in its discretization function.

Furthermore, we have also shown by assuming that all previous subexpressions can be evaluated correctly, we can show that all transitions will also evaluate as long as the type system checks. In other words, given a transition, we proved that if the type can be computed, the operational semantics will also compute. This is a really strong proof since inductively proved all transitions and the base cases, we have shown that our type system is powerful enough to prevent all run-time errors.

5 Type Inference

5.1 Design of our inference engine

Our version of L uses a Hindley-Milner inference engine to perform static type checking at compile time without requiring typing annotations. This gives the programmer the benefit of static type checking at compile time without the encumbrance of having to use type annotations throughout the source code.

The type engine is split up into two components, both of which are run before any portion of the source is ever evaluated but after the lexxing and parsing. The first portion performs constraint gathering by parsing abstract syntax tree and will return the unresolved final type of the program. The second portion takes the constraints that were gathered and performs union-find to solve the system and extract the most generic type of all intermediate expressions as well as the final program.

Furthermore, our type inference engine fully supports all of extensions to the language including polymorphic lists and our stream operators.

5.1.1 Constraint Gathering

In the constraint gathering portion, our implementation performs a recursive descent down the abstract syntax tree generated by the parser to extract type constraints based on how different constructs are used.

The following are the constraints that are gathered for each type of Abstract Syntax Tree node

- `AST_UNOP` We gather a constraint that the type of the parameter must be equivalent to the typeclass of the type of the operator.
- `AST_BINOP` The constraints we gather depends on whether the operation is a cons operator or any other. In the case that it is a cons operator then add a new constraint that the list type of the first parameter must be equal to the type of the second parameter.

  In all other cases it first constraints the type of the first parameter with the type of the second parameter. Then it constraints the type of the of first argument (without loss of generality since both arguments need to have same type) with the type of the operator.

- `AST_BRANCH` Branches have two constraints. First that the type of both branches must be equivalent, and that the type of the predicate must be an Int.
- `AST_EXPRESSION_LIST` Expression lists are the most complicated to handle. An expression list consists of some expression containing a lambda followed by a variable amount of expressions containing the parameters. We recursively process the parameter list, and the last type processed (which in the initial invocation is the arbitrary type of the entire expression list). We start by processing the type of the last parameter, and creating a function type that maps from that type to the last type. It then recurses into the next argument, passing the function type as the last type. Once the argument list has a single element (the function), a constraint is created between the type of the function and the last type.
- `AST_LAMBDA` Lambdas introduce one constraint. That the type of the lambda must be equivalent to a function type which maps the type of the parameter with to the resultant type of the function body.
- `AST_LET` Let's introduce one constraint: that the type of identifier must be equal to the type of the value.
- `AST_LIST` Lists introduce a variable amount of constraints based on the list size. If the list has only one element, it doesn't introduce any constraints. For any lists greater than one element, it creates constraints for every overlapping adjacent pair of elements that their types must be equal.

5.1.2 Constraint Solving

Once all constraints are gathered, our implementation begins the solving process. This is beautifully simple. It simply iterates over all the constraints and unifies them, if the unification fails it reports a relevant error, otherwise it continues on to the next one. If either constraint type is a variable type, it will also check that the variable was defined, to prevent unbound identifier runtime errors.

Once all constraints are solved, it simply returns find of the program's final type. We had to recursively resolve the final type if it was composed of either lists, functions, or weak types and call find on their respective subtypes.

6 Type Inference Examples

6.1 Required Examples

6.2 Example 1

```
let f = lambda x, y. x+y in
let k = (f 2) in
(k 3)
```

```
final solved type:      ConstantType(Int)
5
```

6.3 Example 2

```
let x = lambda a. a in
let a = 1 in
(x 3)
```

```
final solved type:      ConstantType(Int)
3
```

6.4 Example 3

```
let x = lambda y,z. 1+y+z in
let k = (x 4) in
let u = (k 5) in u
```

will yield

```
final solved type:      ConstantType(Int)
10
```

6.5 Example 4

```
let f = lambda x.if x = 0 then 1 else x * (g x)
in
let g = lambda x .(f (x 1))
in
(f 6)
```

```
final solved type:      ConstantType(Int)
720
```

6.6 Example 5

```
fun read_list with n =
    if n = 0 then Nil else
    let val = readInt in
    let rest = (read_list (n 1)) in
    val @ rest
in
let _ = print "Enter number of integers in list" in
let num = readInt in
let _ = print "Please enter the list values:" in
let l = (read_list num) in
let _ = print "List entered" in
let _ = print l in
let _ = print "Adding 2 to each element: " in

fun add with l, n =
    if isNil #l then [!l+n] else
    let hd = !l in
    let tl = #l in
    (hd+n) @ (add tl n)
in

(add l 2)
```

```
final solved type:      List[ConstantType(Int)]
Enter number of integers in list
1
Please enter the list values:
1
List entered
[1]
Adding 2 to each element:
[3]
```

6.7 Example 6

```
fun my_print with x =
  let _ = print "@@" in print x
in
let _ = (my_print "duck") in
let _ = (my_print 7) in
let _ = (my_print [1,2]) in
0
```

```
final solved type:      ConstantType(Int)
@@
duck
@@
7
@@
[1, 2]
0
```

6.8 Example 7

```
let x = 1 @ 2 @ Nil in
let y = if isNil x then 3 else 4 in
y
```

```
final solved type:      ConstantType(Int)
4
```

6.9 Examples of Type System preventing errors

6.9.1 Unbound identifier errors

```
let x = 1 in x + y
```

```
[TYPE ERROR] Found unbound variable: y
From expression: (x + y)}
```

6.9.2 Calling list operators on non-list types

```
!4
```

```
[TYPE ERROR] Constraint unsatisfiable: ConstantType(Int) = List[
   ArbitraryType(1)]
From expression: (!4)}
```

6.9.3 Performing binary operators on two parameters of different types

```
4 + "test"
```

```
[TYPE ERROR] Constraint unsatisfiable: ConstantType(Int) =
   ConstantType(String)
From expression: (4 + "test")}
```

6.9.4 Function application with wrong parameter type for non-polymorphic functions

```
let f = \x.x+1 in (f "string")
```

```
[TYPE ERROR] Constraint unsatisfiable: VariableType(f) = Lambda(
   WeakType(ConstantType(String))[final], ArbitraryType(5))
From expression: (f "string")
```

7 Source code and running

The source code of L is beautifully organized and is all present within the `/src` directory. The source code for the lexer is present in `lexer.l`. The source code for the parser is in `parser.y`. The source code for the interpreter is in `Evaulator.cpp`. The source code for the type inference engine is in `TypeInference.cpp`.

Furthermore, we built L using test-driven development to ensure only the highest quality of code. To run our test-suite execute `./bats test` in the root of the source directory. All test files are located within the `/test` directory. Please note that Dillig's Test 5 requires some input, we recommend just giving `1 {ENTER} 1 {ENTER}` as input.

We hope you enjoy learning about L and using it as much as we did making it.