(g)ROOT
Final Report

Samuel Russo Amy Bui Eliza Encherman
Zachary Goldstein Nickolas Gravel

May 6, 2022

Contents

1 Acknowledgments

Project Mentor: Mert Erden

Professor: Richard Townsend

2 Resources

Implementing functional languages: a tutorial. Simon Peyton Jones, David R Lester. Prentice Hall, 1992.

Modern Compiler Implementation in ML. Andrew W. Appel. Cambridge University Press, 2004.

Compilers: Principles, Techniques, and Tools, 2nd Edition. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Addison-Wesley, 2006.

Engineering a Compiler, 2th Edition. Keith D. Cooper and Linda Torczon. Morgan Kaufmann, 2012.

OCaml Manual

3 Introduction

g(ROOT) is a general-purpose functional programming language, with a pared-down and minimalistic syntax that makes it useful for educational purposes, such as transitioning imperative programmers to functional languages. The language has a LISP-like syntax and offers lambda expressions, higher-order functions, inferred types, and allows variable redefinitions.

g(ROOT) utilizes the Hindley-Milner algorithm in order to infer the types of expressions, as well as typing and variable environments in order to accomplish closure conversion to resolve lambda expressions (closures use the original values of any free variables at the time the closure was created, rather than using the most recent values of any free variables when the closure is used).

The project originally set out to give the language a built-in n-ary tree data structure as a primitive type. This is where the name and mascot ”groot” originates. However, with the challenges of compiling seemingly inherent functional language features and the time constraints, trees are left as a future feature to be implemented.

This document will provide an in-depth language reference manual for g(ROOT), a language tutorial, and details of our implementations and key takeaways from this project.

4 Language Tutorial

4.1 Environment Setup

To run our language, please have the following packages installed:

- `opam-2.1`
- `llvm-13`
- `ocaml-4.13`
- `ocamlbuild-`
- `llc-`
- `gcc-`

If there are compilation issues that may be due to versioning, we have a docker image that may help.

4.2 Compile and Run

*Currently does not support linking, so all source code must be in the same file. gROOT file extension is **.gt***

Compile the g(ROOT) compiler with ONE of these two commands:

- `make`
- `make toplevel.native`

Compile and create executable and intermediate files for any given source file in our language with the format [filename].gt
Intermediate files are stored in toplevel as tmp.ll and tmp.s, respectively

- `make [filename].exe`

Run [filename].exe executable

- `./[filename].exe`

4.3 Using the Language

g(ROOT) uses Lisp-style syntax and allows higher-order functions. All expressions except literals and variable evaluations must be enclosed in parentheses. Whitespace serves no syntactic purpose except to separate tokens and aid in styling.

There is an inbuilt standard basis consisting of the basic algebraic and boolean operations and comparisons, and both named and anonymous user-defined functions can be created using `val lambda` and `lambda`, respectively.

Please reference the g(ROOT) Lexical Conventions for more details.

Simple example code:

```
(val x 42)
(printi x)
```

The above compiled code outputs:

```
$ 42
```

Complex example code:

```
(val letterGrade (lambda (test)
  (if (> 90 test)
      'A'
      (if (> 80 test)
          'B'
          (if (> 70 test)
              'C'
              'D')))))

(printc (letterGrade 89))     (; this prints B ;)

(val computeGrade (lambda (test test2 test3)
  (let ([sum (+ (+ test test2) test3)])
      (let ([avg (/ sum 3)])
          (letterGrade avg)))))

(printc (computeGrade 88 90 91)) (; this prints B ;)

(val letterGrade (lambda (test)
  (if (> 85 test)
      'A'
      (if (> 75 test)
          'B'
          (if (> 65 test)
              'C'
              'D')))))

(printc (letterGrade 89))        (; this prints A ;)
(printc (computeGrade 88 90 91)) (; this prints B ;)
```

The above compiled code outputs:

```
$ B
$ B
$ A
$ B
```

Definitions and expressions are processed in order, and both local and toplevel definitions exist - local definitions take priority in-scope and disappear once the block in which they were defined is finished. Local definitions defined simultaneously cannot reference each other (let* does not exist), but nested let statements can get around this, as seen by using sum to define avg.

Redefinitions are allowed, and later usages use the updated values, though if a variable was used in the body of the closure, the closure will always use that value even if the variable is redefined later, hence why the last line prints “B” - computeGrade still uses the earlier definition of letterGrade.

5 Language Reference Manual

5.1 How to read manual

The syntax of the language will be given in BNF-like notation. Non-terminal symbol will be in italic font *like-this*, square brackets [...] denote optional components, curly braces { ... } denote zero or more repetitions of the enclosed component, and parentheses (...) denote a grouping. Note the font, as `[ ... ]` and `( ... )` are syntax requirements later in the manual.

5.2 Lexical Convention

5.2.1 Blanks

The following characters are considered as **blanks**: space, horizontal tab ('`\t`'), newline character ('`\n`'), and carriage return ('`\r`').

Blanks separate adjacent identifiers, literals, expressions, and keywords. They are otherwise ignored.

5.2.2 Comments

Comments are introduced with two adjact characters `(;` and terminated by two adjacent characters `;)`. Multiline comments are allowed with this. Single line comments using `;;` are also allowed for ease.

```
(; This is a comment. ;)

;; This is another comment

(; This is a
  multi-lined comment. ;)
```

5.2.3 Identifiers

Identifiers are sequences of letters, digits, and ASCII characters, starting with any character that isn't the underscore. Letters will refer to the below ranges of ASCII characters. *Identifiers may not start with an underscore character*, and may not be any of the reserved character sequences.

⟨*ident*⟩ ::= *letter* { *letter* | _ }

⟨*letter*⟩ ::= `!...&` | `*...:` | `<...Z` | `` ` ...z`` | `~` | `|`

5.2.4 Integer Literals

An integer literal is a decimal, represented by a sequence of one or more digits, optionally preceded by a minus sign.

$\langle$*integer-literal*$\rangle$ ::= [`-`] *digit* {*digit*}

$\langle$*digit*$\rangle$ ::= `0...9`

5.2.5 Boolean Literals

Boolean literals are represented by two adjacent characters; the first is the octothorp character (`#`), and it is immediately followed by either the `t` or the `f` character.

$\langle$*boolean-literal*$\rangle$ ::= `#` (`t` | `f`)

5.2.6 Character Literals

Character literals are a single character enclosed by two ' (single-quote) characters.

5.2.7 Operators

All of the following operators are prefix characters or prefixed characters read as single token. Binary operators are expected to be followed by two expressions, unary operators are expected to be followed by one expression.

$\langle$*operator*$\rangle$ ::= (*unary-operator* | *binary-operator*)

$\langle$*unary-operator*$\rangle$ ::= `!` | `-`

$\langle$*binary-operator*$\rangle$ ::= `+` | `-` | `*` | `/` | `mod`
| `==` | $<$ | $>$ | $\leq$ | $\geq$ | `!=`
| `&&` | ||

5.3 Keywords

The below identifiers are reserved keywords and cannot be used except in their capacity as reserve keywords:

```
if      val
let     lambda
```

The following character sequence are also keywords:

```
==      +       &&      >       '
!=      -       ||      mod     #t
<=      *       !       (       #f
>=      /       <       )
```

The following tree-related keywords are still recognized, but their uses are unimplemented. Please be aware: `leaf` `elm` `tree` `cld` `sib`

5.3.1 Syntax

See Definitions and Expression for concrete syntax for each definition and expressions, with detailed examples.

5.4 Values

Base Values

5.4.1 Integer numbers

Integer values are integer numbers in range from -2^{32} to $2^{32}-1$, similar to LLVM's integers, and may support a wider range of integer values on other machines, such as -2^{64} to $2^{64}-1$ on a 64-bit machine.

5.4.2 Boolean values

Booleans have two values. `#t` evaluates to the boolean value `true`, and `#f` evaluates to the boolean value `false`.

5.4.3 Characters

Character values are 8-bit integers between 0 and 255, and follow ASCII standard.

5.4.4 Functions

Functional values are mappings from values to value.

5.5 Types

5.6 Definitions and Expressions

⟨*def*⟩ ::= (`val` *ident* *expr*)
  | *expr*

⟨*expr*⟩ ::= *literal*
  | *ident*
  | *unary-operator* *expr*
  | (*binary-operator* *expr* *expr*)
  | (*ident* {*expr*})
  | (`let` ([*ident* *expr*] {[*ident* *expr*]}) *expr*)
  | (`if` *expr* *expr* *expr*)
  | (`lambda` ({*arguments*}) *expr*)

⟨*literal*⟩ ::= *integer-literal* | *boolean-literal* | *character* | leaf

⟨*arguments*⟩ ::= ϵ
  | *ident* :: *arguments*

Expressions are values or parenthetical expressions.

5.6.1 Values

see Values.

5.6.2 Parenthetical expressions

Parenthetical expressions are always within parentheses and include function application, lambda expressions, global and local definitions, binary and unary operations, and if-statements. In the above concrete syntax, the parentheses in this font, `( ... )`, are syntax requirements, rather than denoting a grouping which is given by (...)

5.6.3 Function application

Function application in (g)ROOT always returns a value, and is written as expression to apply, followed by a list of zero or more expressions, which are its arguments. The arguments are not separated from the applied expression by parentheses. (g)ROOT has first-class functions, therefore functions can be passed as arguments. While partial application is allowed, this feature is use-at-your-own-risk as it may have undefined behavior due to a the recognized bug in type monomorphization.

Example:

```
(foo)
(bar a b)
((baz x) y)
```

5.6.4 Lambda Expression

Lambda expressions are accomplished with the `lambda` keyword, a parentheses-enclosed list of 0 or more identifiers as formal arguments, followed by the expression that may use those arguments and/or any free variables. Nesting of lambda expressions is allowed, but not recommended for the same reason stated above for why partial function application is not recommended.

Example:

```
(lambda () #t)
(lambda (x) x)
(lambda (x y) (+ x y))

(lambda (a) (add2 a b))
```

5.6.5 Global definitions

Global definitions are accomplished using the `val` keyword, followed by an identifier, followed by the expression which is to be bound to that value.

Example:

```
(val x 4)
(val y (+ x 5))
(val foo (lambda (arg) ( * arg arg)))
```

Calling a global definition with a preexisting identifier will re-bind that identifier to the new value - onlys allowed at the top level, and new definition must always of the same type as the previous definition.

5.6.6 Local definitions

Local definitions are found with the `let` expressions, which is the `let` keyword followed by the identifier(s) and the expression(s) to be bound to it, followed by the expression that local variable may be used. Let expressions must have at least one local binding.

Example:

```
(let ([x 4]) (+ 2 x))  (; return 6 ;)
(let ([x 4]) x)        (; return 4 ;)
(let () y)             (; not allowed! ;)
```

Variables defined within the let binding are not defined outside of it, while variables globally relative to the let can be accessed within it. Since let bindings are a type of expression, this allows for chained let bindings.

Example:

```
(let ([x 4])
   (let ([y 5])
      (let ([z 9])
         (+ x (- y z)))))
```

5.6.7 If-expression

If-expressions are the only form of control flow in (g)ROOT, and are always formed with the `if` keyword followed by three expressions (the *condition*, the *true case* and the *false case*). Omission of the false case is a syntax error, and the expressions are not separated by parentheses, brackets, or keywords.

Example:

```
(if #t 1 2)
(if (< 3 4)
   (+ x y)
   (- x y))
```

5.6.8 Unary operators

Unary operations (used for boolean or signed negation) must not be enclosed in parentheses. They are accomplished with a unary operator in front of the expression they negate.

Example:

```
-3
-(+ 3 4)
-(if #t 2 3)
-x

!#t
!x
!(expr)
```

5.6.9 Binary operators

The general use of binary operators is as follows: (*binary-operator* $expr_1$ $expr_2$)

The **arithmetic operators** (`+` , `-` , `*` , `/` , `mod`) take two expressions that evaluate to integers.

The **comparator operators** (`==` , $<$, $>$, $\leq$, $\geq$, `!=`) take two expressions that both evaluate to either integers or booleans.

The **boolean operators** (`&&` , `||`) take two expressions that evaluate to booleans.

5.7 Functions

5.7.1 Built-In Functions

Along with the primitive operators mentioned previously, this language has three built-int print functions:

- `printi`

  Usage: `(printi 42)`

  Purpose: sends the string of an integer to standard out.

- `printc`

  Usage: `(printi ’c’)`

  Purpose: sends the string of an character to standard out.

- `printb`

  Usage: `(printi #t)`

  Purpose: sends the string of an boolean to standard out.

5.7.2 Higher-Order Functions

User-defined functions may be passed as arguments to other functions, and may be returned from functions. Passing around built-in functions as such is not supported.

6 Project Plan

6.1 Planning, Specification, and Development

Initial planning involved going through several different ideas for a language before we settled on a functional language with an achievable feature that stands out. Most members of the group primarily had experience with imperative languages, so this topic provided an interesting challenge for everyone.

We came up with a preliminary syntax for which we could parse and described in our LRM. The syntax was finalized (to include definitions) once parsing of expressions worked and as we moved on to more complex phases of the compiler.

We used GitLab for organization and version control, and relied on clear communication over text, Discord, and in-person to keep each other up-to-date with responsibilities, new tasks, scheduling, road blocks, and goals reached. Every feature was worked on a separate branch, and merges were usually done as a group to resolve merge conflicts between branches together, if any.

Phase 1 (Scanner/Parser) was done as a group. In subsequent phases, different tasks were divided amongsts group members depending on interest, with frequent collaboration with project mentor and other members as the need arises and to double-check each others work. Different responsibilities are described below.

We did not have an official style-guide, and relied on everyone to keep their work readable and well-documented. A few final passes through each file was done to ensure good style.

6.2 Project Timeline

For the most part, we tried to finish as much of a deliverable as possible before a given deadline. For subtasks without any hard due dates, finish-dates became more ambiguous, because a member's work tested by another member of the group, often times, revealed a new bug or a new feature that needs to be implemented before the original task is complete. Due to some of these challenges, we often found ourselves working up to a deadline or working pass an expected finish date.

Goal	**Finish or Submit Date**
Final Language Idea	Feb. 2
Project Proposal	Feb. 4
Phase 1 test script	Feb. 23
Phase 1: Scanner & Parser	Feb. 24
Language Reference Manual	Feb. 28
Final Language Syntax	Mar. 5
Start planning and implementing type-inferencer	Mar. 5
Start planning code generation	Mar. 5
Start planning closure conversion	Mar. 11
Semant for purposes of testing Conversion and Codegen	Mar. 13
Partial Semant and Codegen Module that forces a printing	Mar. 11
New phase 2 test script and reference outputs for all phases	Mar. 24
Phase 2: Hello World	Mar. 28
Conversion	Apr. 15
Codegen	Apr. 19
Extended Test Suite	Apr. 20
Type Inferencer	Apr. 23
Start Monomorphization to deal with infer's polymorphic types	Apr. 23
Debug Conversion & Codegen	Apr. 29
thread primitives through each module	Apr. 29
Debug Infer	May 2
Mono	May 2
Create HAST & Hof (conversion I) phase and fit with Conversion (conversion II) to allow HOFs	May 5
Extend testing script	May 5
Presentation slides	May 5
Presentation	May 6
Final Paper & Submission	May 7

6.3 Roles and Responsibilities

These were the originally assigned recommended roles:

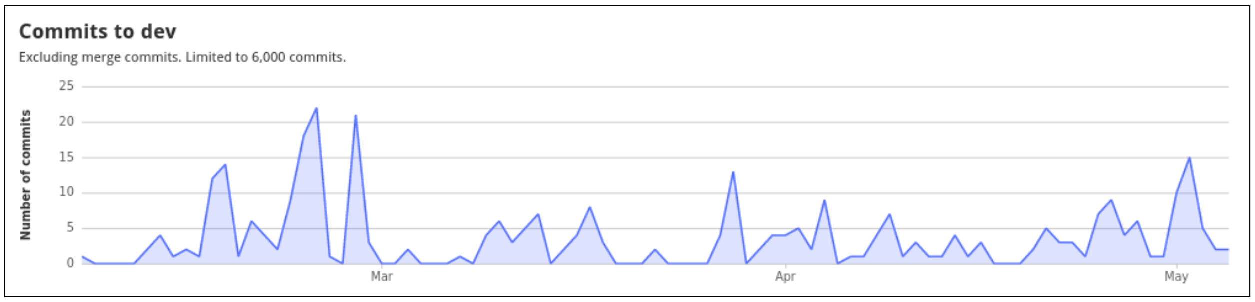
Name	Role
Eliza Encherman	Manager
Sam Russo	Tester
Zachary Goldstein	Language Guru
Nickolas Gravel	System Architect
Amy Bui	Facilitator

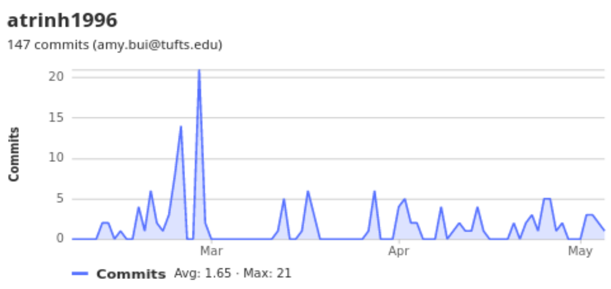
As the project progressed and implementing features of a functional language became more difficult, we abandoned these roles and each subgroup or member focused on progressively implementing different language features. Everyone was responsible intially testing their own feature and subsequent debugging, but we were also responsible for checking each others work and providing “fresh eyes” when testing someone else’s implemented feature. Here is a list of each person’s primary responsibility:

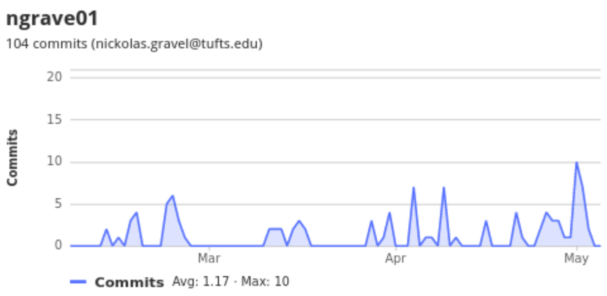
Name	Responsibilities
Eliza Encherman	Primitive recognition and testing in Scanner/Parser, Scope, Infer, Conversion, and primitive code generation
Sam Russo	Type Inferencer
Zachary Goldstein	Errors & Warnings, test suite, Docker environment
Nickolas Gravel	Type Inferencer
Amy Bui	monomorphizer, conversion I & II, codegen

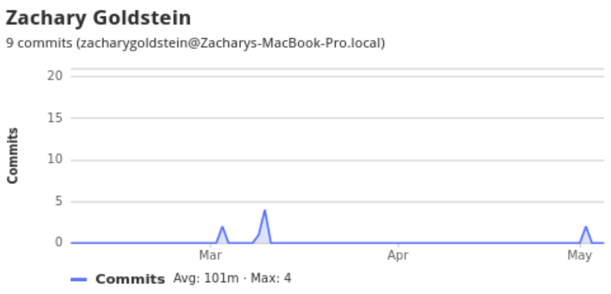
6.4 Development Tools

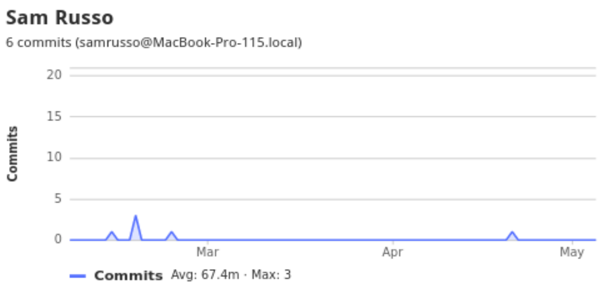
6.5 Project Logs

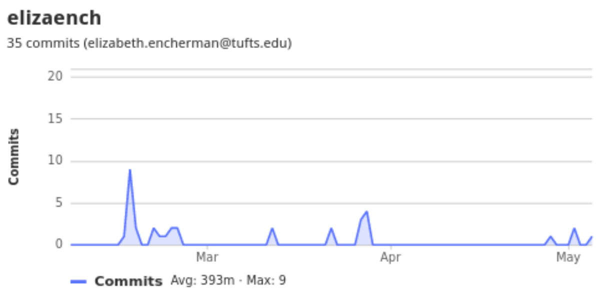




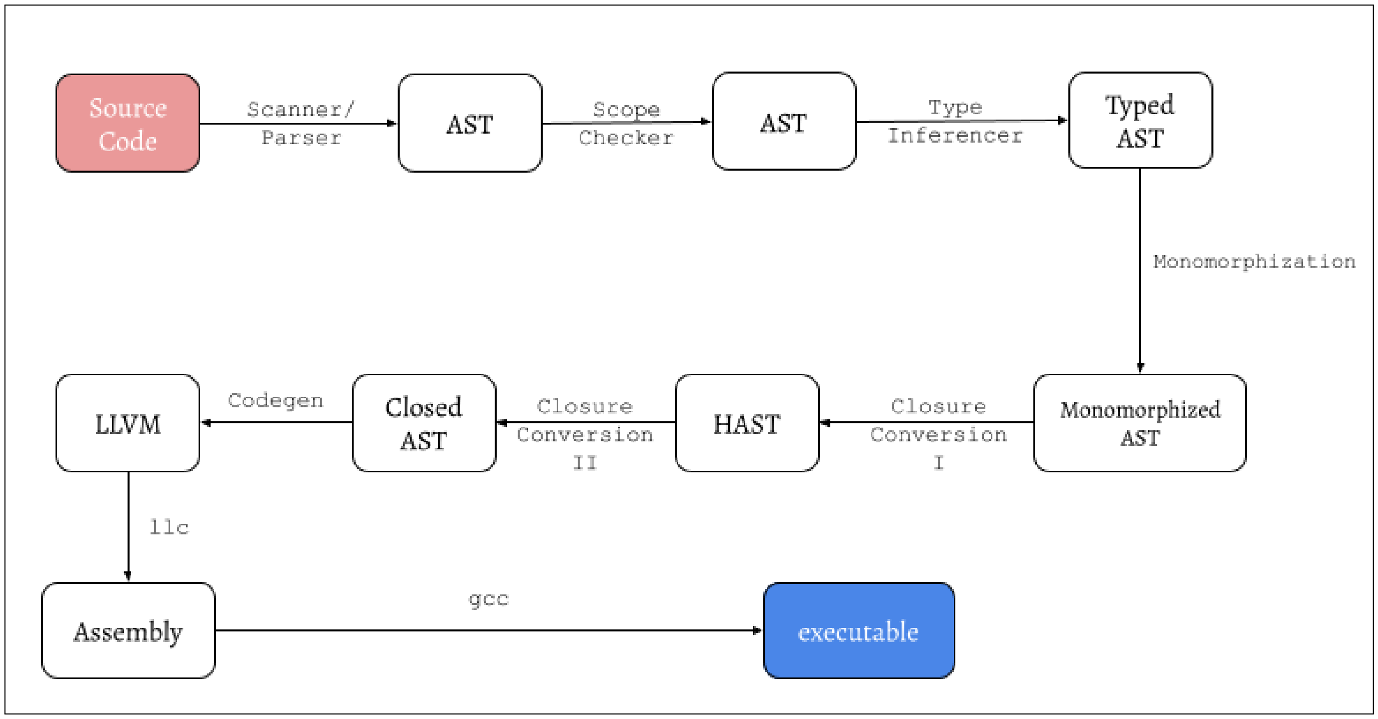








7 Architectural Design



7.1 Scanner/Parser

- Given the groot source code, the Scanner/Parser scans and tokenizes sequences of characters to build the basic abstract syntax tree of the program.
- *Authored by the whole group. Final syntax incorporated into Scanner/Parser done by Zach.*

7.2 Scope Checker

- This module performs a partial semantic check on a given abstract syntax tree, in which any variable name used has been bound and is used in the scope that it is allowed. The module reports an unbound variable, if any; otherwise, the same abstract syntax tree it receives is outputted.
- *Authored by Amy, with help incorporating primitives from Eliza.*

7.3 Type Inferencer

- The type inference takes an abstract syntax tree, and assumes all named variables are bound to some value. (g)ROOT is implicitly typed, so this module detects the types of all expressions and definitions. We follow the Hindley-Milner type system method in order to infer types. Using the types already given by primitives and literals, the inferencer generates constraints and then solves those constraints in order to deduce the type of the current definition it is inferring; those constraints are then used to infer the type of the rest of the program. The type inferencer will throw an error if it catches any type errors, such as type mismatches in function application of arguments. The module allows for polymorphism, so a definition may be typed to a type variable; however, to make resolving polymorphic types easier, this module also disallows nested lambdas. The inferencer outputs the final typed abstract syntax tree, or TAST.

- *Authored by Sam and Nik, with help from Mert. Primitives were incorporated by Eliza.*

7.4 Monomorphization

- This module takes a typed abstract syntax tree, which may or may not have any type variables, and monomorphizes it, getting rid of any polymorphism such that the tree has concrete types for everything. This is accomplished by tagging expressions that are polymorphic, finding where they are used (application), and using the types of the arguments it is used with to match type variables to concrete types; a mono-typed definition of the original polymorphic variable is then inserted into the program for that specific use case. The module outputs a monomorphic-typed abstract syntax tree.
- Bug: The algorithm we implemented for this resolves polymorphic types for non-nested lambda definitions, but cannot resolve them for nested lambdas which involves partial function application in order to get concrete types for each type variables. We concluded additional passes to resolve polymorphic types was required, but ran out of time for this. So, to ensure no type variables leak through to subsequent phases, Mono does one additional pass through the program, re-typing any leftover type variables to our int type. It is a recognized bug, so nested lambdas may result in undefined behavior or llc will raise an error regarding any type mismatches.
- *Authored by Amy.*

7.5 Closure Conversion I (Hof)

- Conversion I (or Hof) takes a monomorphic-typed abstract syntax tree, and creates partial closures for every lambda expression by re-typing every lambda expression from a function type to a new abstract type we called a closure type. Conversion I's closure types will carry a unique name to associate with the lambda expression (generated in the module), the original lambda's return type, the list of the formal types, AND a list of the types of the free variables. This is considered "partial" because no closure thus far is closed with the values or a reference to the values of the free variables.

  Re-typing in this module helps subsequent modules deal with higher-order functions more easily, because Conversion I finds uses of a Hof, and re-types the function type to the appropriate closure type in function application

  This module returns a h-abstract syntax tree. As indicated by the name, this intermediate phase makes handling higher-order functions easier in the next module.

- *Authored by Amy.*

7.6 Closure Conversion II (Conversion)

- Conversion II takes a h-abstract syntax tree, and finishes creating closures for every lambda expression. The new closure type in this module now caries the unique name previously described in Conversion I appended with a string to indicate it is referring t othe type of the lambda, the lambda's function type augmented to include the types of the free variables appended to the list of the formal types, and a separate list of just the types of the free variables, if any. A closed lambda expression itself caries the original unique name, which will link it to its function definition later, and a list of the free variables converted into typed variable expressions. The closed lambda no longer carries the expression representing the body of the lambda, because this module creates a record type that represents a function defintion for the lambda, which does carry the lambda's body expression and has a function name that matches the name of the lambda. These function definitions are set aside in a separate list to be passed to the next phase. Also being passed to the next phase is a list of all closure types created.

  The other significant task this module has is to track the number of times a named variable is redefined; this allows closures to use the original value of the free variable it was closed with even after that variable gets redefined elsewhere.

  The final output of this module is a struct containing the closed abstract syntax tree, list of function definitions, list of closure types generated during this phase, and a map of names to a list of their occurrences and types.

- *Authored by Amy, with help from Mert, and with help incorporating primitives from Eliza.*

7.7 Code Generation

- Code generation produces the llvm instruction for each definition and expression in the program. Each kind of definition and expression produce a particular pattern of llvm instructions. Given the information described in Closure Conversion II, code generation has four major steps to accomplish this:

  1. Every "closure type" mentioned previously gets declared a named struct type in the llvm (using the available function type and free variable types); they all have the following common structure:

     $\{$ `fptr* ; ` $\{$ `typ ; ` $\}$ $\}$

     Where the first struct member is always a pointer to the function created as a lambda's function definition, and all subsequent member(s), if any, are where the values of free variables will get stored.

  2. Every named variable (including versions of it, should it have been defined multiple times) is globally defined and initialized to zero or nullptr depending on whether or not it is a function type, struct type, or some constant. When global

   variables are referred to in the program, it is either to assign them a value or use them. Because all definitons are evaluated in order in a main function (discussed next), no variable is used before a value is stored in them in the llvm.

   3. Every definition/expression in the closed ast gets turned into llvm instructions and put in a "main" function body. This allows us to mimic sequential execution.

   4. Every "function definition" mentioned previously is declared and defined by generating llvm instruction for the body's expression.

- *Authored by Amy and Eliza.*

8 Test Plan

9 Lessons Learned

9.1 Amy Bui

My main takeaway is that compiler writing, or even writing just a single optimization for a compiler, is a very complicated endeavor. We did not end up with a lot of our original goals and features we intended for our functional language, but the detours we took while exploring this whole other paradigm was very enriching, and I was never bored and always challenged. Everyone should write a compiler at least once before graduating. This was a great project in demonstrating the practical application of the concepts and theory we learnt in 170 and 105, and I'd recommend taking both classes before 107 or at the same time for a more comfortable time. I have a lot of appreciation for people who do research and contribute to the field of functional languages, and the complexity of lambda calculus. My advice to future students is that you should have a burning desire to implement type inferencing and lambda calculus before you settle on a a functional language as your project topic; I never knew how much we took static typing and statment blocks for granted. Or you can get a thrill from just taking on this challenge.

9.2 Nickolas Gravel

Compiler writing is a long, long journey filled with various pitfalls, unexpected challenges, dragons, and possibly avocados. For myself, I found the learning curve to be slightly steeper than my peers. Before this class, I had not taken CS105 Programming Languages or had any experience coding in a functional language. So, in the beginning, I encountered some tribulation understanding these concepts. Effectively, I not only needed to learn the basics of compiler writing, but needed to learn the basic components of a programming language, and the basics of coding in a functional language as well.

Then came type inferencing, a module that we had greatly underestimated the amount of time and work it would take to implement. I was one of the main programmers that worked on implementing this module. Again, I found that having some previous experience with type inferencing would have been a huge help here...but we persevered! I dove into the Hadley-Milner type system algorithm, and with the guidance of Mert and Michael Ryan Clarkson, after about a month of coding the and a week of debugging we had a working type inferencer!

All in all, building a compiler is no joke. Even seemingly trivial steps in implementing a compiler were found to be not so trivial, requiring us to brain storm creative solutions to get to the next step. For anyone building a compiler, I recommend spending ample time writing up a thorough design. Having a written design indicating which data structures and data types were being passed and returned from each module would have helped in overall organization and may have prevented some of the various bugs that we encountered. Future students, if you are set on building a language with inferred types make sure you have a strong understanding how to implement type inferencing and design your inferencer thoroughly before implementing it.

Designing a programming language and building a compiler for it was riveting experience. I learned so much about functional languages, type inferencing, and what it really takes to compile source code down to a final executable. Everyone with any interest in low-level programming should take this course, but I would recommend taking a course in programming languages more gradual learning curve.

9.3 Sam Russo

There are so many independent aspects to working on a compiler, from language design/syntax to memory to warnings to additional features. Coming up with solutions to the problems that we found in each of these domains and dealing with bugs was challenging, but it was also hard to ensure that the choices we made in each of the domains worked well with each other. If one person decided to do thing x while working in one area, sometimes we would realize later that that decision was incompatible or poorly compatible with a decision someone else made in a different domain at the same time.

Working with a group as big as ours for as long as we did (and I know in the scheme of things our team wasn't that big and we weren't working together for that long) is really hard. Challenges were mostly accountability (definitely including my own) and stemmed from everyone's having different schedules and different workloads. Different workloads meant that people 1) prioritized our project different amounts, 2) had differing amounts of time to dedicate to it, and 3) had different ideas of how to spend the time when we worked on it together. Specifically, the grad students in our group had much more time to work on the project, which led to pretty different levels of contribution. For future mixed grad-undergrad groups, I would think about this before committing to such a group, and if people decide to stick with it, then they should be very clear around group expectations.

9.4 Eliza Encherman

I learned a great deal about languages and compilers from this project - I definitely agree with Amy that everyone should write a compiler before graduating, and I'd recommend the language creation step as well, as I feel like I really understand the code I write and see better, as well as understand some of the thought processes behind some the peculiarities of different languages. I also whole-heartedly agree with my groupmates about the difficulties involved with inplementing functional languages and type inferencing - this project very much reinforced how what is simpler for the user is often far more complicated to implement than something like static typing, that's a little more work on the user's side but much easier to check.

I also would emphasize the importance of good practices like documentation, throwing specific error statements, and making functions to print structures as you introduce structures - I frequently would be spending hours in debugging just finding where a specific error was coming from, which would then often be followed by hours more because it was hard to understand what a function did or what a variable was at a certain moment in time.

9.5 Zachary Goldstein

- Takeaway:
- Advice: