g(root)

Amy Bui
Eliza Encherman
Zach Goldstein
Sam Russo
Nickolas Gravel

start

1 Sam

Table of Contents

2 next

2 Sam

Introduction

3 next

3 Sam

g(root) Key Features

Functional Language
General-purpose functional language, All expressions return a value.

Lisp-like Syntax
Fully parenthesized pre-fixed notation.

Inferred Types
Utilizes the Hadley-Milner type system algorithm to infer Int, Bool, and Char types

Higher-Order Functions
Functions can be *passed* and *returned* from functions.

Variable Redefinition
Top-level variable redefinition updates all previous references to the redefined value

4 next

4 Sam

"Finally a functional language the whole galaxy can enjoy!"

- Rocket Raccoon

5 next

5 Sam

Motivations...in the beginning.

A ***functional*** language that is ***intuitive*** and ***easy to use*** for first time functional programmers.

Minimalistic programming language — we only give you what you need, no bells and whistles.

Tree data structure as a **basic primitive type.**

6 next

6 Sam

But things change...

A ***functional*** language that is ***intuitive*** and ***easy to use*** for first time functional programmers.

Minimalistic programming language — we only give you what you need, no bells and whistles.

~~***Tree*** data structures as a **basic primitive type.**~~

7 next

7 Sam

g(root) Tutorial

8 next

8 Zach

(g)root grammar

```
⟨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
```


9 Zach

Code Snippets

```
1 (val x 4)
2 (print x)
3                              (; prints 4 ;)
4 (val x (+ 2 x))
5 (printi (x))                 (; prints 6 ;)
6
7 (printi if(> 0 x) x ~x)      (; prints -6 ;)
```

Assign *x* a value, modify the *x*, and print the *x*

- **Lisp-style** syntax
- Sequential Execution
- All expressions are literals and variables enclosed in parentheses
- **Redefinition** is allowed

10 Zach

Code Snippets

```
1 (val year 2021)
2 (print year)                 (; prints 2021 ;)
3
4 (val myYear (lambda () year))
5 (printi (myYear))            (; prints 2021 ;)
6
7 (val year 2022)
8 (printi (myYear))            (; prints 2021 ;)
```

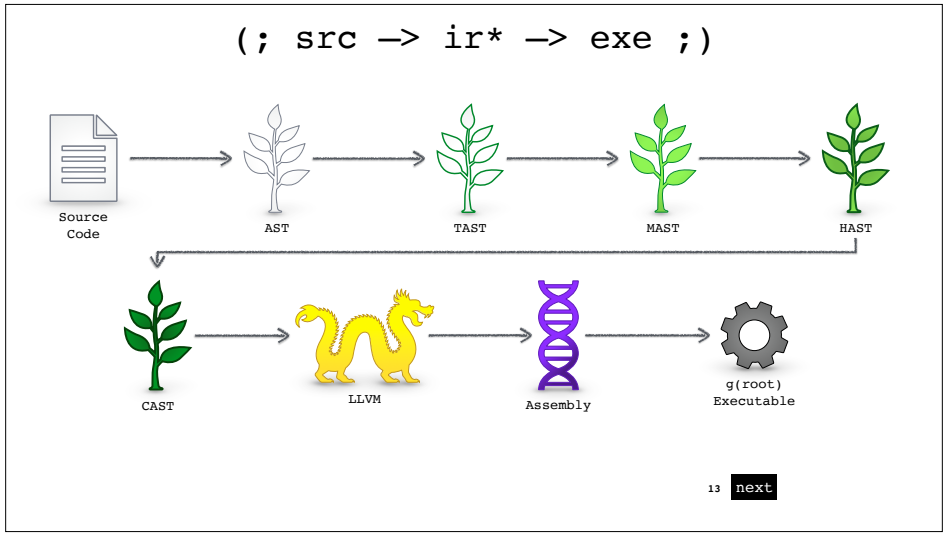
Supports both named and anonymous functions using lambdas.

- Variable values remain **fixed** inside the closure at the time the closure was made
- **Redefinition** of the variable is still allowed

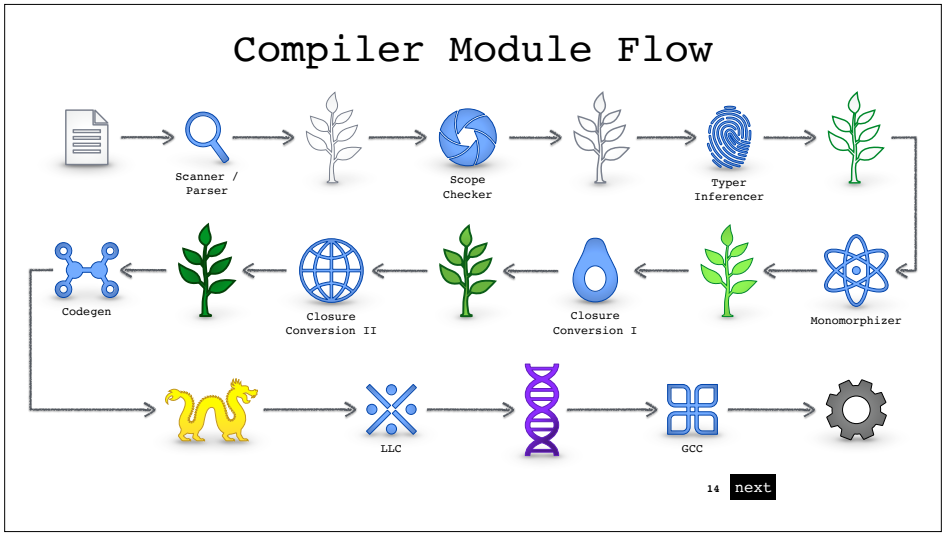
11 Zach

Compiler Architecture

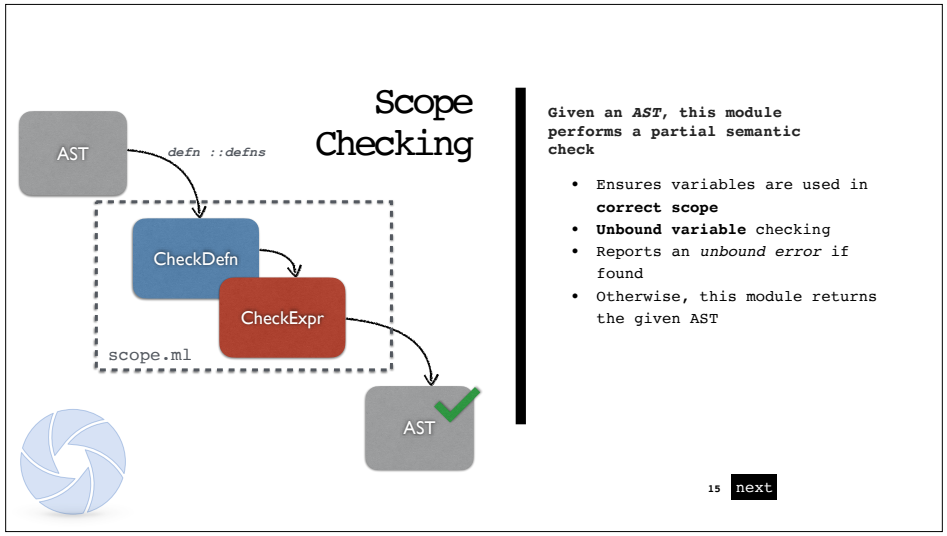
12 Nick



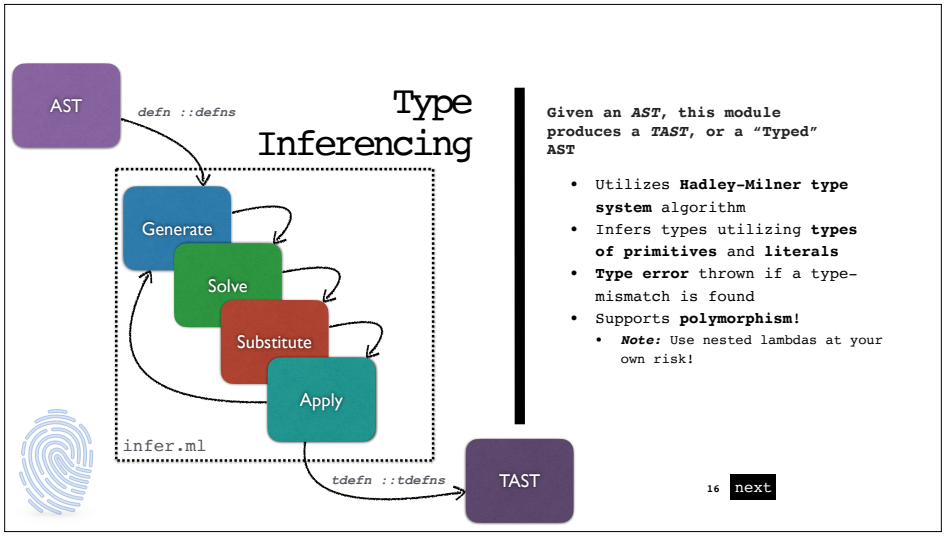
13 Nick



14 Nick



15 Nick



16 Nick / Sam

Monomorphization

```
1 (; retn : 'a -> 'a ;)
2 (val retn (lambda (n) n))
3
4 (retn 13)   (; int ;)
5
6 (retn #f)   (; bool ;)
7
8 (retn 'c') (; char ;)
```

```
1  (; retn : int -> int ;)
2  (val retn (lambda (n) n))
3  (retn 13)
4
5  (; retn : bool -> bool ;)
6  (val retn (lambda (n) n))
7  (retn #f)
8
9  (; retn : char -> char ;)
10 (val retn (lambda (n) n))
11 (retn 'c')
```

Given an *TAST*, this module produces a *MAST*, or a "Monomorphized" AST

- Searches the TAST for any use of polymorphic expressions
- Injects a *mono-typed* definition to resolve all polymorphism
- **Bugs**:
  - **Int type** is inserted for any expression still bound to a type variable

Closure Conversion I

```
1 let partial_closure_type(id, retty, formaltys, freetys) =
2    HTycon (HCls (id, retty, formaltys, freetys))
3
```

Given an *MAST*, this module produces a *HAST*, or "H for H" AST

- ***Partial* Closure Conversion for Lambda Expressions**
  - Re-types lambda expressions to newly defined **closure type**
  - **Unique name** - indicates a function call to execute, or which closure type to use
  - **Types** of **return**, **formals**, and **free variables**
- Detects uses of **higher-order functions**, and augments the types and expressions to refer to new closure type.

Closure Conversion II

```
1 let function_definition =
2 {
3   bodyExpression;
4   returnType;
5   functionName;
6   formals;
7   freeVars;
8 }
9
```

Given an *HAST*, this module produces a *CAST*, or a "Closed" AST

- Redefined variables are tracked
- A **new function definition** is created for every lambda expression
- The new definition carries the lambda's body

Closure Conversion II

```
1 closure _lambda0 {
2   int (int, int)* ;       (;_func0;)
3 }                         (; retn ;)
4
5 int _func0 (int n, int m) {
6   return n;
7 }
```

```
1 ...
2 (val retn (lambda (n m) n))
3 (val x 42)
4 ...
5
6
7
8 ... lambda (a) (retn x a) ...
```

```
1  closure _lambda1 {
2    int (int, _lambda0*, int)* ;   (;_func1;)
3    _lambda0* ;
4    int;
5  }
6
7  int _func1 (int a, _lambda0* retn, int x) {
8    func = retn[0];
9    result = func(x a);
10   return result;
11 }
```

Putting the lid on lambda expressions.

- **Closed Lambda Expression**
  - *Unique name* - indicates function call to execute correct lambda body
  - *List of Free Variables - as Var expressions*
    - Names are modified with most recent occurrence number appended
- **Closure Type of Closed Lambdas:**
  - List of the types of the free variables
  - Augmented function type

Code Generation

Int	i32
Bool	i1
Char	i8*
Closures	unique, named struct ptrs: { fptr; [frees] }

abstract types become ***lltypes***

Given a *CAST* this module translates expressions into *LLVM*

- Every *definition* is an *instruction* in main()
- Each lambda's body expression is a **uniquely** named function definition
- Top-level val definitions assign values to **global variable names**
- **Redefinitions** allowed by tagging variable with the **occurrence #**
  - This ensures closures use the **correct value of a free variable** even if that variable gets redefined

21 Eliza

Conclusions

22 Nick

Writing a type inferencer took away a lot of resources.

Two out of our five members spent much of the time implementing this module. It took a lot of time and resources to complete.

Lesson learned:
Implement a working compiler without inferred typing, *then* introduce type inferencing.

23 Nick

Give g(root) roots, and branches, and leaves!

Initially, we planned to implement a primitive tree ADT for g(root). However, due to time, we decided to save this feature for later release.

Keep an eye out for **g(root) vol. 2**!

24 Nick

Start with an interpreter when implementing a functional language,

Implementing an interpreter early in development would have helped generate test cases, and flesh out ideas early in the development process.

25 next

25 Nick

Demonstration

26 next

26 Eliza

FIN.

Presenters:
Amy Bui
Eliza Encherman
Zach Goldstein
Sam Russo
Nickolas Gravel

close

27