Language Proposal: *tail*

‘A tail recursion optimization language’ -Some Wise Gal

Team Roles

Team Members	**UNIs**	**Roles**
Sandra Shaefer Jennifer Lam Serena Shah-Simpson Fiona Rowan	sns2153 jl3953 ss4354 fmr2112	Project Manager Systems Architect Tester Language Guru

Language Description

Our language aims to optimize recursion through tail-call optimization on the compiler level. Our compiler will eliminate overhead incurred by tail recursive algorithms by bypassing the allocation of new stack frames within recursive function calls. The main idea is to allocate constant stack space for each recursive function, making recursive functions more time and space efficient on average. Our language syntax also explicitly encourages users to conceptualize recursion as a set of base cases and repetition rules.

Motivation

Tail recursion is a useful aspect of structured programming, but in many general purpose languages, it is less efficient than its iterative counterpart. Implementing tail recursion using Java syntax, for instance, requires using control flow and loops with conditions, general concepts used for implementing non-recursive functions as well, and therefore the syntax does not lend itself well to distinguishing between computing tail recursive functions vs. other kinds of functions in the backend.

We wish to leverage the standard form of tail recursive functions in order to make them easier to conceptualize in code and more efficient to run. Tail calls are subroutine calls executed at the last line of a function, and tail recursion is when the subroutine call is the function itself. For such functions, we wish to allocate constant stack space, replacing function stack frames by those of subsequent subroutine calls instead of allocating space for new stack frames on the call stack. We drew inspiration for our tail call elimination language from features of Scheme and some tail call optimizing C compiler options.

Summary of Goals

- A high-level language for optimizing tail recursion by reducing stack overhead
- Utilize intuitive syntax to improve readability and accessibility of recursive functions
- Compile to LLVM for its speed and optimization infrastructure

Domain features

- Constant stack allocation for tail-recursive function calls
- Syntax and function typing to make recursive and iterative functions easier to write and understand
- Utilize LLVM to communicate with lower levels of control

Language Design

Our language will eliminate the overhead incurred by tail recursion at the compiler level. We will bypass the need to allocate new stack frames for subroutines in LLVM. By the time a subroutine is called at the end of a function scope, much of the stack frame of the current function call is useless, so instead of allocating memory for a new stack on the call stack, we will pass information down to the subroutine's stack and replace the current stack with the subroutine's stack. At the compiler level, the program can jump to the subroutine call instead of allocating space for a new stack. Our language syntax will lends itself well to conceptualizing recursive algorithms.

Code Syntax

Built-In Types

int	A sequence of digits that don't contain decimals.
float	A sequence of digits that do contain decimals.
char	A sequence of characters between quotes.
bool	A boolean type that can evaluate to either true or false.
int[]	An array of ints.
float[]	An array of floats.
char[]	An array of chars.

bool[]	An array of booleans.
null	A special value that indicates an uninitialized variable.

Function Syntax: <**functiontype> <Name> <arg> … <arg> { … }**

Function definitions will start with one of three function type words, followed by the name of the function. Following the name of the function are any number of arguments, separated by spaces. After the arguments, the braces denote the beginning of the body of the function. User must specify a return statement at the end of each function scope, which consists of a colon (":") followed by the value to be returned. Recursive and main functions have predefined return statements.

Function Types

recursive	Designates a function that will use recursive logic. This function type must include the **base_case** and **rec** keywords.
iterative	Designates a function that uses iteration. This function type must include the **rep** keyword.
declarative	Designates an declarative function. Declarative functions don't have keywords, they execute code sequentially as one would expect from a traditional programming language such as C or Java.

Control Flow

<condition> : [<action>]	Colon denotes a case. Conditional to be checked at the left of colon. Action (or list of commands) to be performed at the right. Brackets **[]** optional if there is only one command (highly recommended). **<conditional> : <action>** syntax can be listed (with **default**) as the last case. See code examples. Drawn from OCaml's if-then-else and pattern-matching concepts.
default : [<action>]	Default case for a list of conditionals, must come at the end of every list of conditions (even if there is only one condition). Can be empty or just null.

Comments

(* I am a comment *)	Same comment syntax for single-line and multi-line comments.
(* I am a (*	Allow nested comments with same syntax

nested*) comment *)	

Keywords

Keywords are separated into categories according to the function type.

init	Defines the initialization of variables to be used in any function. Variables cannot be defined without this keyword.
base_case	Defines the base case to be approached in recursive functions. Also defines return values (optional) after each case, as designated by a colon (“:”).
rec	Designates the portion of code that will be called recursively in a *recursive* function. This must include a return, designated by a colon, that calls the function name.
rep	Designates the portion of code that will be called repeatedly in an *iterative* function.

Operators

We support + - * / % for integers, floats, and chars. Precedence is taken to be the standard arithmetic order. We also support the boolean operators AND as ‘&&’ and OR as ‘||’.

Scoping

Scoping of variables is taken to be standard c scoping (i.e. between **{ }** or **[ ]**, which we call enclosures). Variables defined within curly braces or brackets are visible only inside. Variables defined outside can be modified inside. A variable declared between enclosures that shares a name with variable of more global scope (declared outside, but visible inside) is the one that is used.

Code Samples

```
(* Factorial function in recursive form *)
recursive Factorial_R n {
     base_case {
          n == 1 : 1
     }
     rec {
```

```
            n * Factorial_R n-1
      }
}

(* Factorial function in iterative form *)
iterative Factorial_I n {
      init {
            i = 1
            prod = 1
      }
      rep n {
            prod = prod * i
            i = i + 1
      }
      : prod
}

(*    Recursively perform a binary search on an array A
      Return value of position if n found, -1 otherwise *)
recursive Binary_Search n lo hi A {
      init {
            mid = (lo + hi) / 2
      }
      base_case{
            lo > hi: -1
            A[mid] == n: mid
      }
      rec{
            A[mid] < n: Binary_Search n mid+1 hi
            A[mid] >= n: Binary_Search n lo mid-1
      }
}

(* Declarative program example. Simply prints "Hello World" *)
declarative Hello_World {
      print "Hello World"
      :null
}

(* Declarative program demonstrating control-flow *)
```

```
declarative Control_Flow ice_cream {

     (* Uses single-command style until “default” case for
     illustration purposes *)
     ice_cream == “mango” : print “fave”
     ice_cream == “chocolate” : print “allergic”
     ice_cream == “taiwanese McDonald’s”: print “Edwards recommends”
     default: [
          a = “Ice cream is overrated”
          print a
     ]
     :null
}

declarative main n {
     A = [2,5,7,1,4,8]
     Hello_World
     Factorial_R 5
     Factorial_I 5
     Binary_Search 7 0 length(A)-1 A
     Control_Flow “taiwanese McDonald’s”
}
```