

Biu: a Static-typed Functional Language and Its Compiler

杨至轩 张天宇

The BiU Programming Language

Primitive Types

- Bool
- Char
- Number: currently represented by 64-bit float number

Examples:

```
(define pi 3.141592653569)
(define x true)
(define new-line '\n')
```

Compound Types

- Function type: $(\Rightarrow \text{Arguments} \cdots \text{Result})$
- Array type: $(\text{Array ElementType})$

Examples:

```
;; type of transform: ( $\Rightarrow$  ( $\Rightarrow$  Number Number) Number Number)
(define (transform
        (f ( $\Rightarrow$  Number Number))
        (x Number))
  (f x))

;; type of array: (Array Bool)
(define array (make-array Bool 100))
```

Functions

- Functions can be nestedly defined
- Variables are lexically scoped
- Functions can be passed as values (as closures)

Examples:

```
(define (derivative (f (=> Number Number)))  
  (define delta 0.000001)  
  (define (d (x Number))  
    (/ (- (f (+ x delta))  
          (f (- x delta)))  
        (+ delta delta)))  
    ;; f is visible in d,  
    ;; even after returning from derivative  
  d)  
  
  ;; poly: x^2 + x  
  (define (poly (x Number))  
    (+ x (* x x)))  
  
  ;; its numerical derivative  
  (define poly-d (derivative poly))  
  
  (print (poly-d 5))  
  ;; 11.00000
```

Type Check

A type checked program is:

- **progress**: in a step (of compiling or execution), the program has a legal instruction
- **preservation**: after every step, the type of every variable stay unchanged

Type Check

Biu is static-typed:

- Programmers are required to write *type annotations*
- Type checking is performed after parsing. Programs failed in type checking will be *rejected*
- *Efficient code* can be generated with type information

Type Checking Examples

```
(define (derivative (f (=> Number Number)))
  (define delta 0.000001)
  (define (d (x Number))
    (/ (- (f (+ x delta))
          (f (- x delta)))
        (+ delta delta)))
  ;; f is visible in d,
  ;; even after returning from derivative
  d)

;; poly: x^2 + x
(define (poly (x Number))
  (+ x (* x x)))

;; its numerical derivative
(define poly-d (derivative poly))

(print (poly-d 5))
;; 11.00000

;;;;;;;;;;;;;;

$ ./biuc <tests/diff.biu
Biu type of delta : Number
Biu type of d : (Number) -> (Number)
Biu type of derivative : ((Number) -> (Number)) -> ((Number) -> (Number))
Biu type of poly : (Number) -> (Number)
Biu type of poly-d : (Number) -> (Number)
finished typechecking
```


Language Library

Biu programs can **link** with C/C++ programs, which makes exporting C functions to Biu easy:

```
typedef struct closureType{
    void *func;
    void *env;
} closure;

static double printnumber_func(void* env, double a)
{
    double ret = (double)printf("%f\n", a);
    fflush(stdout);
    return ret;
}
closure print = {printnumber_func, (void*)0};

;;; in Biu:
(extern-raw print (=> Number Number) "print")
(print 5)
```

Demo

a Brainf**k interpreter written in Biu

biuc

a biu compiler under the LLVM framework

Overview

- **Lexer & Parser:** a hand written LL(1) parser
- **Type checking:** syntax-directed type checker
- **Code generation:** syntax-directed translator targeting to LLVM IR
- **Compiling & Linking:** LLVM tool chain

Lexer

The lexer converts the code text into a token stream of:

- char literal: 'a'
- bool literal: true, false
- number literal: 2.7182818
- symbol: define, if, set!, ...
- other characters: (,)

Lexer Example

```
if(isdigit(lastChar) || lastChar == '-') {  
    // number: -?[0-9]+(.[0-9]+)?  
    string numStr = "";  
    do {  
        numStr += lastChar;  
        lastChar = getchar();  
    } while(isdigit(lastChar) || lastChar == '.');  
  
    if(numStr == "-") {  
        symbolStr = "-";  
        return tok_symbol;  
    }  
  
    const char* endpoint;  
    numVal = strtod(numStr.c_str(), nullptr);  
    return tok_number;  
}
```

Parser

The parser parses the token stream into a tree of AST nodes:

- NumberAST, SymbolAST, ...
- DefineVarFormAST
- DefineFuncFormAST
- ApplicationFormAST
- IfFormAST
- And other special forms

Parser Example

```
unique_ptr parseForm()
{
    if(curTok != '(') {
        parserError(string("parseForm expects a '(', get: ") + tok2str(curTok));
        return nullptr;
    }
    getNextToken();

    if(curTok == tok_symbol && symbolStr == "if") {
        // if-form
        getNextToken();
        auto form = llvm::make_unique();
        form->condition = parseExpr();
        form->branch_true = parseExpr();
        if(curTok != ')') {
            form->branch_false = parseExpr();
        }

        if(curTok != ')') {
            parserError(string("parseForm: if-form expect a ')', get: ") + tok2str(curTok));
            return nullptr;
        }
        getNextToken();
        return std::move(form);
    } else {
        // parse other special forms ....
    }
}
```


Code Generation

Main problem:

1. How to compile nested functions in the **low level** target language (LLVM IR, Assembly or C)
2. How to compile closures (a inner function can access variables defined in outside function even if outside functions have **returned**)

Flat Closures

1. Free variables of a function are transformed into a **environment** argument of this function
2. Nested functions are compiled into **global** (with different names)
3. When a function in biu is defined, the environment for this function is created in heap. And the function variable is a pair of a **pointer to code** and a **pointer to the environment**

Flat Closures Example

```
(define (factory (x Number))  
  (define (g)  
    x)  
  g)
```

```
typedef struct closureType{  
    void *func;  
    void *env;  
} closure;  
  
typedef __eng_g{  
    double x;  
} env_g;  
  
double g_func(env_g* e)  
{  
    return e->x;  
}  
  
closure factory_func(double x)  
{  
    env_g* env = malloc(sizeof(env_g));  
    env->x = x;  
  
    closure g_cls;  
    g_cls.func = g_func;  
    g_cls.env = env;  
  
    return g_cls;  
}
```

Optimization & Assembling

- Currently, no optimization is performed
- We use LLVM's tool to generate assembly code and object code

Currently Missing

- The most important missing feature of Biu is *polymorphism*
- *Parametric* polymorphism (like OCaml/Haskell and template of C++) or *sub-typing* polymorphism (like class derivation of C++ and Java) can be considered
- Another important missing feature is user defined type

Tucaao

- The main problem about LLVM is its incompleteness of documentation
- Type systems make both compiler writers and programs writers easier

很惭愧，就做了一点微小的工作，
谢谢大家