CALC Language Compiler

Wiktor Szydłowski (75135)
Valerii Matviiv (75176)

1. Introduction

CALC is a functional-imperative language with static typing, first-class functions, closures, and mutable references. The compiler generates LLVM IR with runtime support for memory management and closures.

2. Type System

2.1 Type Definitions

```
type calc_type =
  | IntT | BoolT | UnitT | StringT
  | RefT of calc_type
  | FunT of calc_type * calc_type
  | TTuple of calc_type list
  | TRecord of (string * calc_type) list
  | TList of calc_type
```

2.2 Positive Test: Arithmetic

Source programs/function.calc:

```
let double = fun (x: int) -> x * 2 in
let result = double(21) in
printInt(result);
printEndLine()
```

Type: `double: int -> int, result: int`

LLVM Output:

```
define i32 @lambda_1(ptr %0, i32 %1) {
entry:
  %2 = mul nsw i32 %1, 2
  ret i32 %2
}
define i32 @main() #0 {
L0:
  %0 = bitcast ptr @lambda_1 to ptr
  %1 = call ptr @create_closure(ptr %0, ptr null)
  %2 = call i32 @apply_closure_i32_i32(ptr %1, i32 21)
  call void @print_int(i32 %2)
  call void @print_endline()
  ret i32 0
}
```

Output: `42`

2.3 Negative Tests

Type errors caught at compile-time:

```
5 + true                    (* Typing error: Expecting Integer *)
fun (x: int) -> x(5)        (* Error: Unknown type *)
let x = new(5) in x := true (* Typing error: Type mismatch in assignment *)
if true then 5 else false   (* Typing error: Branch must have same type *)
```

Implementation:

```
let type_int_int_int_bin_op mk e1 e2 =
  match type_of e1, type_of e2 with
  | IntT, IntT -> mk IntT e1 e2
  | _ -> mk (None "Expecting Integer") e1 e2
```

3. Control Flow

3.1 Conditionals

```
let x = 5 in
if x > 0 then
  printInt(x)
else
  printInt(0 - x);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = icmp sgt i32 5, 0
  br i1 %0, label %L1, label %L2
L1:
  call void @print_int(i32 5)
  br label %L3
L2:
  %1 = sub nsw i32 0, 5
  call void @print_int(i32 %1)
  call void @print_endline()
  br label %L3
L3:
  %2 = phi i32 [0, %L1], [0, %L2]
  ret i32 0
}
```

Output: `5`

3.2 While Loops

Source programs/factorial.calc:

```
let n = new(5) in
let result = new(1) in
while !n > 0 do (
  result := !result * !n;
  n := !n - 1
);
printInt(!result);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = call ptr @new_ref_int(i32 5)
  %1 = call ptr @new_ref_int(i32 1)
  br label %L1
L1:
  %2 = call i32 @deref_int(ptr %0)
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %L2, label %L3
L2:
  %4 = call i32 @deref_int(ptr %1)
  %5 = call i32 @deref_int(ptr %0)
  %6 = mul nsw i32 %4, %5
  call void @assign_int(ptr %1, i32 %6)
  %7 = call i32 @deref_int(ptr %0)
  %8 = sub nsw i32 %7, 1
  call void @assign_int(ptr %0, i32 %8)
  br label %L1
L3:
  %9 = call i32 @deref_int(ptr %1)
  call void @print_int(i32 %9)
  call void @print_endline()
  ret i32 0
}
```

Output: `120`

4. Data Structures

4.1 Tuples

Source programs/simple_tuple.calc:

```
let t1 = (10, 20, 30) in
printInt(t1.0);
printInt(t1.1);
printInt(t1.2);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = getelementptr {i32, i32, i32}, ptr null, i32 1
  %1 = ptrtoint ptr %0 to i32
  %2 = call ptr @malloc(i32 %1)
  %3 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 0
  store i32 10, ptr %3
  %4 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 1
  store i32 20, ptr %4
  %5 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 2
  store i32 30, ptr %5
  %6 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 0
  %7 = load i32, ptr %6
  call void @print_int(i32 %7)
  %8 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 1
  %9 = load i32, ptr %8
  call void @print_int(i32 %9)
  %10 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 2
  %11 = load i32, ptr %10
  call void @print_int(i32 %11)
  call void @print_endline()
  ret i32 0
}
```

Output: 102030

Type: `(int * int * int)` → LLVM `{i32, i32, i32}` on heap

4.2 Records

Source programs/simple_record.calc:

```
let r1 = {x = 10; y = 20; z = 30} in
printInt(r1.x);
printInt(r1.y);
printInt(r1.z);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = getelementptr {i32, i32, i32}, ptr null, i32 1
  %1 = ptrtoint ptr %0 to i32
  %2 = call ptr @malloc(i32 %1)
  %3 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 0
  store i32 10, ptr %3
  %4 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 1
  store i32 20, ptr %4
  %5 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 2
  store i32 30, ptr %5
  %6 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 0
  %7 = load i32, ptr %6
  call void @print_int(i32 %7)
  %8 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 1
  %9 = load i32, ptr %8
  call void @print_int(i32 %9)
  %10 = getelementptr {i32, i32, i32}, ptr %2, i32 0, i32 2
  %11 = load i32, ptr %10
  call void @print_int(i32 %11)
  call void @print_endline()
  ret i32 0
}
```

Output: `102030`

Type: `{x:int; y:int; z:int}` → Fields indexed by sorted position

4.3 Lists

Source programs/simple_list.calc:

```
let l1 = [10, 20, 30, 40, 50] in
printInt(l1[0]);
printInt(l1[4]);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = getelementptr i32, ptr null, i32 5
  %1 = ptrtoint ptr %0 to i32
  %2 = call ptr @malloc(i32 %1)
  %3 = getelementptr i32, ptr %2, i32 0
  store i32 10, ptr %3
  %4 = getelementptr i32, ptr %2, i32 1
  store i32 20, ptr %4
  %5 = getelementptr i32, ptr %2, i32 2
  store i32 30, ptr %5
  %6 = getelementptr i32, ptr %2, i32 3
  store i32 40, ptr %6
  %7 = getelementptr i32, ptr %2, i32 4
  store i32 50, ptr %7
  %8 = getelementptr i32, ptr %2, i32 0
  %9 = load i32, ptr %8
  call void @print_int(i32 %9)
  %10 = getelementptr i32, ptr %2, i32 4
  %11 = load i32, ptr %10
  call void @print_int(i32 %11)
  call void @print_endline()
  ret i32 0
}
```

Output: `1050`

Type: `[int]` → Contiguous array on heap

5. Memory Management

5.1 References

Source programs/mutation.calc:

```
let counter = new(0) in
counter := !counter + 1;
counter := !counter + 1;
counter := !counter + 1;
printInt(!counter);
printEndLine()
```

LLVM Output:

```
define i32 @main() #0 {
L0:
  %0 = call ptr @new_ref_int(i32 0)
  %1 = call i32 @deref_int(ptr %0)
  %2 = add nsw i32 %1, 1
  call void @assign_int(ptr %0, i32 %2)
  %3 = call i32 @deref_int(ptr %0)
  %4 = add nsw i32 %3, 1
  call void @assign_int(ptr %0, i32 %4)
  %5 = call i32 @deref_int(ptr %0)
  %6 = add nsw i32 %5, 1
  call void @assign_int(ptr %0, i32 %6)
  %7 = call i32 @deref_int(ptr %0)
  call void @print_int(i32 %7)
  call void @print_endline()
  ret i32 0
}
```

Output: `3`

6. Functions and Closures

6.1 Simple Closure

Source:

```
let make_adder = fun (x: int) -> (
  fun (y: int) -> x + y
) in
let add10 = make_adder(10) in
printInt(add10(5));
printEndLine()
```

LLVM Output:

```
define i32 @lambda_2(ptr %0, i32 %1) {
entry:
  %2 = getelementptr {i32}, ptr %0, i32 0, i32 0
  %3 = load i32, ptr %2
  %4 = add nsw i32 %3, %1
  ret i32 %4
}

define ptr @lambda_1(ptr %0, i32 %1) {
entry:
  %2 = getelementptr {i32}, ptr null, i32 1
  %3 = ptrtoint ptr %2 to i32
  %4 = call ptr @malloc(i32 %3)
  %5 = getelementptr {i32}, ptr %4, i32 0, i32 0
  store i32 %1, ptr %5
  %6 = bitcast ptr @lambda_2 to ptr
  %7 = call ptr @create_closure(ptr %6, ptr %4)
  ret ptr %7
}
define i32 @main() #0 {
L0:
  %0 = bitcast ptr @lambda_1 to ptr
  %1 = call ptr @create_closure(ptr %0, ptr null)
  %2 = call ptr @apply_closure_i32_ptr(ptr %1, i32 10)
  %3 = call i32 @apply_closure_i32_i32(ptr %2, i32 5)
  call void @print_int(i32 %3)
  call void @print_endline()
  ret i32 0
}
```

Output: 15

6.2 Stateful Closure

Source:

```
let make_counter = fun (init: int) -> (
  let count = new(init) in
  fun (inc: int) -> (
    count := !count + inc;
    !count
  )
)
in
let counter = make_counter(0) in
printInt(counter(1));
printInt(counter(2));
printInt(counter(3));
printEndLine()
```

LLVM Output:

```
define i32 @lambda_2(ptr %0, i32 %1) {
entry:
  %2 = getelementptr {ptr}, ptr %0, i32 0, i32 0
  %3 = load ptr, ptr %2
  %4 = call i32 @deref_int(ptr %3)
  %5 = add nsw i32 %4, %1
  call void @assign_int(ptr %3, i32 %5)
  %6 = call i32 @deref_int(ptr %3)
  ret i32 %6
}

define ptr @lambda_1(ptr %0, i32 %1) {
entry:
  %2 = call ptr @new_ref_int(i32 %1)
  %3 = getelementptr {ptr}, ptr null, i32 1
  %4 = ptrtoint ptr %3 to i32
  %5 = call ptr @malloc(i32 %4)
  %6 = getelementptr {ptr}, ptr %5, i32 0, i32 0
  store ptr %2, ptr %6
  %7 = bitcast ptr @lambda_2 to ptr
  %8 = call ptr @create_closure(ptr %7, ptr %5)
  ret ptr %8
}
define i32 @main() #0 {
```

```
L0:
  %0 = bitcast ptr @lambda_1 to ptr
  %1 = call ptr @create_closure(ptr %0, ptr null)
  %2 = call ptr @apply_closure_i32_ptr(ptr %1, i32 0)
  %3 = call i32 @apply_closure_i32_i32(ptr %2, i32 1)
  call void @print_int(i32 %3)
  %4 = call i32 @apply_closure_i32_i32(ptr %2, i32 2)
  call void @print_int(i32 %4)
  %5 = call i32 @apply_closure_i32_i32(ptr %2, i32 3)
  call void @print_int(i32 %5)
  call void @print_endline()
  ret i32 0
}
```

Output: `136`

7. Optimization

7.1 Constant Folding

Source programs/optimization_test.calc:

```
let x = 10 + 20 in
let y = (if true then 5 else 100) in
printInt(x + y);
printEndLine()
```

Without Optimization (`make run FILE=optimization_test.calc`):

```
define i32 @main() #0 {
L0:
  %0 = add nsw i32 10, 20
  br i1 1, label %L1, label %L2
L1:
  br label %L3
L2:
  br label %L3
L3:
```

```
  %1 = phi i32 [5, %L1], [100, %L2]
  %2 = add nsw i32 %0, %1
  call void @print_int(i32 %2)
  call void @print_endline()
  ret i32 0
}
```

With Optimization (`make run FILE=optimization_test.calc OPT=1`):

```
define i32 @main() #0 {
L0:
  call void @print_int(i32 35)
  call void @print_endline()
  ret i32 0
}
```

Output: `35`

Implementation (optimizer.ml):

```
let rec optimize env e =
  match e with
  | Add (e1, e2) ->
      (match optimize env e1, optimize env e2 with
       | Num n1, Num n2 -> Num (n1 + n2)
       | o1, o2 -> Add (o1, o2))
  | If (e1, e2, e3) ->
      (match optimize env e1 with
       | Bool true -> optimize env e2
       | Bool false -> optimize env e3
       | o1 -> If (o1, optimize env e2, optimize env e3))
```

8. Complete Example

Source:

```
let make_adder = fun (x: int) -> (
  fun (y: int) -> x + y
) in

let fib_iter = fun (n: int) -> (
  if n <= 1 then n
  else (
    let a = new(0) in
    let b = new(1) in
    let i = new(1) in
    while !i < n do (
      let temp = !a + !b in
      a := !b;
      b := temp;
      i := !i + 1
    );
    !b
  )
) in

printString("--- START ---");
printEndLine();

printString("Curried Adder: ");
let add10 = make_adder(10) in
printInt(add10(5));
printEndLine();

printString("Fibonacci Loop: ");
let idx = new(0) in
while !idx < 6 do (
  let f = fib_iter(!idx) in
  printInt(!idx);
  printString(":");
  printInt(f);
  printString("  ");
  idx := !idx + 1
);
printEndLine();
```

```
let pair_val = (100, 200) in
printString("Tuple Access: ");
printInt(pair_val.0 + pair_val.1);
printEndLine();

let list_val = [1, 2, 3, 4] in
printString("List Access: ");
printInt(list_val[3]);
printEndLine();

let rec_val = {id = 50; val = 60} in
printString("Record Access: ");
printInt(rec_val.id + rec_val.val);
printEndLine();

printString("--- DONE ---");
printEndLine()
```

LLVM Output:

```
define i32 @lambda_2(ptr %0, i32 %1) {
entry:
  %2 = getelementptr {i32}, ptr %0, i32 0, i32 0
  %3 = load i32, ptr %2
  %4 = add nsw i32 %3, %1
  ret i32 %4
}

define ptr @lambda_1(ptr %0, i32 %1) {
entry:
  %2 = getelementptr {i32}, ptr null, i32 1
  %3 = ptrtoint ptr %2 to i32
  %4 = call ptr @malloc(i32 %3)
  %5 = getelementptr {i32}, ptr %4, i32 0, i32 0
  store i32 %1, ptr %5
  %6 = bitcast ptr @lambda_2 to ptr
  %7 = call ptr @create_closure(ptr %6, ptr %4)
  ret ptr %7
}

define i32 @lambda_3(ptr %0, i32 %1) {
entry:
```

```
  %2 = icmp sle i32 %1, 1
  br i1 %2, label %L1, label %L2
L1:
  br label %L3
L2:
  %3 = call ptr @new_ref_int(i32 0)
  %4 = call ptr @new_ref_int(i32 1)
  %5 = call ptr @new_ref_int(i32 1)
  br label %L4
L4:
  %6 = call i32 @deref_int(ptr %5)
  %7 = icmp slt i32 %6, %1
  br i1 %7, label %L5, label %L6
L5:
  %8 = call i32 @deref_int(ptr %3)
  %9 = call i32 @deref_int(ptr %4)
  %10 = add nsw i32 %8, %9
  %11 = call i32 @deref_int(ptr %4)
  call void @assign_int(ptr %3, i32 %11)
  call void @assign_int(ptr %4, i32 %10)
  %12 = call i32 @deref_int(ptr %5)
  %13 = add nsw i32 %12, 1
  call void @assign_int(ptr %5, i32 %13)
  br label %L4
L6:
  %14 = call i32 @deref_int(ptr %4)
  br label %L3
L3:
  %15 = phi i32 [%1, %L1], [%14, %L6]
  ret i32 %15
}
define i32 @main() #0 {
L0:
  %0 = bitcast ptr @lambda_1 to ptr
  %1 = call ptr @create_closure(ptr %0, ptr null)
  %2 = bitcast ptr @lambda_3 to ptr
  %3 = call ptr @create_closure(ptr %2, ptr null)
  %5 = bitcast ptr @.str.3 to ptr
  call void @print_string(ptr %5)
  call void @print_endline()
  %7 = bitcast ptr @.str.5 to ptr
  call void @print_string(ptr %7)
  %8 = call ptr @apply_closure_i32_ptr(ptr %1, i32 10)
```

```
  %9 = call i32 @apply_closure_i32_i32(ptr %8, i32 5)
  call void @print_int(i32 %9)
  call void @print_endline()
  %11 = bitcast ptr @.str.9 to ptr
  call void @print_string(ptr %11)
  %12 = call ptr @new_ref_int(i32 0)
  br label %L1
L1:
  %13 = call i32 @deref_int(ptr %12)
  %14 = icmp slt i32 %13, 6
  br i1 %14, label %L2, label %L3
L2:
  %15 = call i32 @deref_int(ptr %12)
  %16 = call i32 @apply_closure_i32_i32(ptr %3, i32 %15)
  %17 = call i32 @deref_int(ptr %12)
  call void @print_int(i32 %17)
  %19 = bitcast ptr @.str.17 to ptr
  call void @print_string(ptr %19)
  call void @print_int(i32 %16)
  %21 = bitcast ptr @.str.19 to ptr
  call void @print_string(ptr %21)
  %22 = call i32 @deref_int(ptr %12)
  %23 = add nsw i32 %22, 1
  call void @assign_int(ptr %12, i32 %23)
  br label %L1
L3:
  call void @print_endline()
  %24 = getelementptr {i32, i32}, ptr null, i32 1
  %25 = ptrtoint ptr %24 to i32
  %26 = call ptr @malloc(i32 %25)
  %27 = getelementptr {i32, i32}, ptr %26, i32 0, i32 0
  store i32 100, ptr %27
  %28 = getelementptr {i32, i32}, ptr %26, i32 0, i32 1
  store i32 200, ptr %28
  %30 = bitcast ptr @.str.28 to ptr
  call void @print_string(ptr %30)
  %31 = getelementptr {i32, i32}, ptr %26, i32 0, i32 0
  %32 = load i32, ptr %31
  %33 = getelementptr {i32, i32}, ptr %26, i32 0, i32 1
  %34 = load i32, ptr %33
  %35 = add nsw i32 %32, %34
  call void @print_int(i32 %35)
  call void @print_endline()
```

```
  %36 = getelementptr i32, ptr null, i32 4
  %37 = ptrtoint ptr %36 to i32
  %38 = call ptr @malloc(i32 %37)
  %39 = getelementptr i32, ptr %38, i32 0
  store i32 1, ptr %39
  %40 = getelementptr i32, ptr %38, i32 1
  store i32 2, ptr %40
  %41 = getelementptr i32, ptr %38, i32 2
  store i32 3, ptr %41
  %42 = getelementptr i32, ptr %38, i32 3
  store i32 4, ptr %42
  %44 = bitcast ptr @.str.42 to ptr
  call void @print_string(ptr %44)
  %45 = getelementptr i32, ptr %38, i32 3
  %46 = load i32, ptr %45
  call void @print_int(i32 %46)
  call void @print_endline()
  %47 = getelementptr {i32, i32}, ptr null, i32 1
  %48 = ptrtoint ptr %47 to i32
  %49 = call ptr @malloc(i32 %48)
  %50 = getelementptr {i32, i32}, ptr %49, i32 0, i32 0
  store i32 50, ptr %50
  %51 = getelementptr {i32, i32}, ptr %49, i32 0, i32 1
  store i32 60, ptr %51
  %53 = bitcast ptr @.str.51 to ptr
  call void @print_string(ptr %53)
  %54 = getelementptr {i32, i32}, ptr %49, i32 0, i32 0
  %55 = load i32, ptr %54
  %56 = getelementptr {i32, i32}, ptr %49, i32 0, i32 1
  %57 = load i32, ptr %56
  %58 = add nsw i32 %55, %57
  call void @print_int(i32 %58)
  call void @print_endline()
  %60 = bitcast ptr @.str.58 to ptr
  call void @print_string(ptr %60)
  call void @print_endline()
  ret i32 0
}
```

Output:

```
--- START ---
Curried Adder: 15
Fibonacci Loop: 0:0  1:1  2:1  3:2  4:3  5:5
Tuple Access: 300
List Access: 4
Record Access: 110
--- DONE ---
```

9. Conclusion

Key Achievements

Type System: Static checking prevents type errors at compile-time with clear error messages.
Closures: First-class functions with environment capture via heap-allocated structures.
Memory: Type-safe mutable references through runtime system with type-specific operations.
LLVM: Code generation to LLVM IR language.
Optimization: Constant folding and propagation eliminate runtime computation.

Technical Highlights

1. Environment-based type checking with scoped variable bindings
2. Free variable analysis identifies captured variables for closures
3. Type-specific runtime ensures safety with separate functions per type
4. Heap allocation for all compound data structures (tuples, records, lists)
5. Parallel let bindings evaluated in same scope

10. Project Structure

```
src/
├── ast.ml, lexer.mll, parser.mly  # Front-end
├── typing.ml, env.ml              # Type system
├── eval.ml, mem.ml                # Interpreter
├── optimizer.ml                   # Optimization
├── llvm.ml                        # Code generation
├── mem_runtime.c/h                # Memory runtime
├── closure_runtime.c/h            # Closure runtime
└── calc.ml, calcc.ml              # Entry points
```

Thank you

"There are only two kinds of languages:
the ones people complain about and the ones nobody uses."
Bjarne Stroustrup