Implementing Continuation based language in LLVM and Clang

Kaito TOKUMORI
University of the Ryukyus
Email: kaito@cr.ie.u-ryukyu.ac.jp

Shinji KONO
University of the Ryukyus
Email: kono@ie.u-ryukyu.ac.jp

Abstract—**A programming paradigm which use data segments and code segments is proposed. CbC is a lower language of C for this paradigm. CbC has standalone compiler and GCC version. In this study, we add an implement CbC compiler on LLVM/clang-3.7. The detail of implementation and evaluation are shown.**

I. A Practical Continuation based Language

We proposed units of program named code segment and data segment. Code segment is a unit of calculation which has no state. Data segment is a set of typed data. Code segments are connected to data segments with a context, which is a meta data segment. After an execution of a code segment and its context, next code segments (Continuation) is executed.

We had developed a programming language "Continuation based C (CbC)" [1]. Hearafter we call it CbC, which supports code segments. CbC is compatible with the C language and it has continuation as a goto statement.

Code segments and data segments are low level enough to represent computation details, and it is architecture independent. It can be used as an architecture independent assembler.

CbC has standalone compiler and GCC version. Here we report new partial implementation of CbC compiler based on LLVM and Clang 3.7.

First we show the CbC language overview.

II. Continuation based C

CbC's basic programing unit is a code segment. It is not a subroutine, but it looks like a function, because it has input and output. These interfaces should be data segments and we are currently designing data segments part.

```
1  __code f(Allocate allocate){
2    allocate.size = 0;
3    goto g(allocate);
4  }
5
6  // data segment definition
7  // (generated automatically)
8  union Data {
9    struct Allocate {
10     long size;
11   } allocate;
12 };
```

TABLE I
CbC Example

In this example, a code segment **f** has input datasegment **allocate** (Allocate is a data segment identifier) and sends output it to a code segment **g**. CbC compiler generates data segment definition automatically so we do not have to write it. There is no return from code segment **g**, **g** should call another continuation using **goto**. Code segments has one input data segment and several output data segment, and their dependency is proved by data segments.

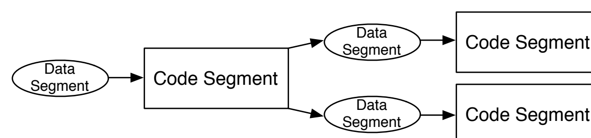

Fig. 1. Code Segments and Data Segments on CbC

In CbC, we can go to a code segment from a C function and we can call C functions in a code segment. So we don't have to shift completely from C to CbC. The later one is straight forward, but the former one needs further extensions.

```
1  int main() {
2    goto hello("Hello␣World\n", __return,
     __environment);
3  }
4
5  __code hello(char *s, __code(*ret)(int, void*),
     void *env) {
6    printf(s);
7    goto (*ret)(0);
8  }
```

TABLE II
Call C Functions in a Code Segment

In this hello world example, the environment of **main**() and its continuation is kept in a variable **__environment**. The environment and the continuation can be get using **__environment** and **__return**.Arbitrary mixture of code segments and functions are allowed. The continuation of **goto** statement never returns to original function, but it goes to caller of original function. In this case, it returns result 0 to the operating system. This cotinuation is called **goto with environment**.

III. LLVM and Clang

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies, and the LLVM Core libraries provide a modern source and target independent optimizer, along with code generation support for many popular

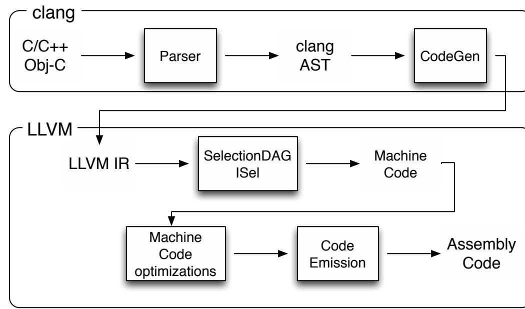

Fig. 2. LLVM and Clang structure

CPUs. Clang is an LLVM native C/C++/Objective-C compiler. Figure 2 is Clang and LLVM's compile flow.

LLVM has a intermediate representation which called LLVM IR[2]. Importantly, we do not modify it so we do not have to modifiy optimize part.

IV. Implementation in LLVM and Clang

So how to implement CbC compiler in LLVM and Clang. Here we show our idea.

- Code segments are implemented by C functions.
- Transition is implemented by forced tail call.
- Goto with environment is implemented by setjmp and longjmp.

__code is implemented as a new type keyword in LLVM and Clang. You may think **__code** is an attribute of a function, which means that the function can call in tail call elimination only. Because of this implementation, we can actually call code segment as a C function call.

Forcing tail call require many conditions. For example, there should be no statement after tail call, caller and callee's calling convention have to be the same and type is cc10, cc11 or fastcc, callee's return value type have to be the same as caller's it, add tail call elimination pass, and so on.

All code segment has the void return type and we do not allow to write statement after continuation, so type problem and after statement problem is solved.

Tail call elimination pass is enabled in **BackendUtil.cpp**. In the clang, when optimize level is two or more, tail call elimination pass is enabled. We modify it to enable anytime and if optimize level is one or less, tail call elimination pass work for only code segment.

Next, we solve a calling convention problem. We select fastcc for code segment's calling convention. In the clang, calling convention is managed by CGFunctionInfo class and its infomations are set in **CGCall.cpp**. We modify here to set fastcc to code segments.

Goto with environment is implemented by code rearranging. If the **__environment** or **__return** is declared, CbC compiler rearrange code for goto with environment. We use setjmp and longjmp for it. Setjmp save environment before continuation, and longjmp restore it.

V. Result

Here is our benchmark program.

```
1  int f0(int i) {
2    int k,j;
3    k = 3+i;
4    j = g0(i+3);
5    return k+4+j;
6  }
7
8  int g0(int i) {
9    return h0(i+4)+i;
10 }
11
12 int h0(int i) {
13   return i+4;
14 }
```

TABLE III
BENCHMARK PROGRAM CONV1

It is written in C and CbC and there are several optimization is possible. When argument is 1, use CbC continuation. When argument is 2 or 3, optimization is enabled.

Here we show benchmark result (TABLE IV).

	./conv1 1	./conv1 2	./conv1 3
Micro-C	6.875	2.4562	3.105
GCC -O2	2.9438	0.955	1.265
LLVM/clang -O0	5.835	4.1887	5.0625
LLVM/clang -O2	3.3875	2.29	2.5087

TABLE IV
EXECUTION TIME(S)

LLVM and clang compiler is faster than Micro-C when optimization is enabled. It mean LLVM's optimization is powerful and useful. LLVM and clang compiler is slower than GCC but GCC cannot compile safety without optimization. It means LLVM can compile more reliability than GCC.

VI. Conclusion

We have designed and implemented Continuation based language for practical use. We have partial implementation of CbC using LLVM and Clang 3.7. CbC can use LLVM's optimization. We did not modify LLVM IR to implement CbC compiler.

In future, we design and implement data segment and meta code segment, meta data segment for meta computation.

References

[1] S. Kono and K. Yogi, "Implementing continuation based language in GCC," *CoRR*, vol. abs/1109.4048, 2011. [Online]. Available: http://arxiv.org/abs/1109.4048
[2] LLVM Language Reference Manual, *http://llvm.org/docs/LangRef.html*.
[3] LLVM Documentation, *http://llvm.org/docs/index.html*.
[4] clang documentation, *http://clang.llvm.org/docs/index.html*.