Compiling Functional Programs to C0 Bytecode

Oguz Ulgen
Carnegie Mellon University
oulgen@andrew.cmu.edu

ABSTRACT
Hi, this is my abstract.

General Terms
Theory, Languages, Compilers

Keywords
Language definitions, functional programming, C0, bytecode

1. INTRODUCTION
Compilers are written to convert a source language to a target language. For practical purposes, the target language is always either a binary form that is referred as object code or an intermediary language that can than recompiled into a binary form. The first case is straight forward since the conversion is done from the source into machine readable format. However, the latter case is not as simple. In most applications, compiler designers chose to compile down to an intermediate language in order to use a highly optimized compiler to generate the machine readable binary form. This intermediate language is generally `LLVM` language or assembly language.

In this paper, we chose to use `C0` bytecode as our intermediate language. It is possible to read about this language on `http://c0.typesafety.net/`.

Another choice we have made is to compile a functional language, specifically a PCF (Programming Computable Functions) language.

2. STACK BASED IMPLEMENTATION OF AN INTERPRETER FOR THE *CLAC* LANGUAGE
Before moving on to implementing a compiler for the PCF language, we chose to write an interpreter for the *CLAC* language. The reason for making this decision stems from the fact that both *CLAC* language and `C0` bytecode have a stack based procedure for evaluation.

```
Clac ::= Num Int
       | Op Op
       | Pair
       | Prj1
       | Prj2
       | If
       | Skip

Op ::= Add | Sub | Mult | Div
```

Figure 1: The *CLAC* language definition

Before Stack		Before Queue		After Stack		After Queue
S	||	n, Q	$\longrightarrow$	S, n	||	Q
S, x, y	||	`+`$, Q$	$\longrightarrow$	$S, x + y$	||	Q
S, x, y	||	`-`$, Q$	$\longrightarrow$	$S, x - y$	||	Q
S, x, y	||	`*`$, Q$	$\longrightarrow$	$S, x * y$	||	Q
S, x, y	||	`/`$, Q$	$\longrightarrow$	$S, x \,/\, y$	||	Q
S, x, y	||	`Pair`$, Q$	$\longrightarrow$	$S, \langle x, y\rangle$	||	Q
$S, \langle x, y\rangle$	||	`Prj1`$, Q$	$\longrightarrow$	S, x	||	Q
$S, \langle x, y\rangle$	||	`Prj2`$, Q$	$\longrightarrow$	S, y	||	Q
$S, 0$	||	`If`$, tok_1, tok_2, Q$	$\longrightarrow$	S, tok_1	||	Q
$S, 1$	||	`If`$, tok_1, tok_2, Q$	$\longrightarrow$	S, tok_2	||	Q
S, n	||	`Skip`$, Q$	$\longrightarrow$	S	||	$Q[n : end]$

Figure 2: Stack/queue based Clac reference

The resemblance between *CLAC* language and `C0` bytecode enables us to consider `C0` bytecode at a more abstract way by just looking at a significantly smaller language that is *CLAC* language.

3. COMPILATION TO *C0* BYTECODE
In order to discuss the process of compilation of PCF to `C0` bytecode, we must first formally define the PCF language.

3.1 Language Constructors
In our representation of PCF language, we have used the following primitives: `void`, `unit` and `nat`. From these primitives, we have constructed sum and product types which we are shown as `pair` and `sum`. These primitives and constructed types enabled us to implement `projections` and `injections`. Finally, through these high level constructs,

Exp e ::=		
	$\mathtt{z}$	$\mathtt{z}$
	$\mathtt{s}(e)$	$\mathtt{s}(e)$
	$\mathtt{ifz}(e; e_0; x.e_1)$	$\mathtt{ifz}\ e\ \{z \Rightarrow e_0 \mid \mathtt{s}(x) \Rightarrow e_1\}$
	$\mathtt{lam}[\tau](x.e)$	$\mathtt{fn}\,(x:\tau)\,e$
	$\mathtt{let}(e_1; x.e_2)$	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$
	$\mathtt{pair}(e_1; e_2)$	$\langle e_1, e_2 \rangle$
	$\mathtt{pr}[\mathtt{l}](e)$	$e \cdot \mathtt{l}$
	$\mathtt{pr}[\mathtt{r}](e)$	$e \cdot \mathtt{r}$
	$\mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e)$	$\mathtt{inl}[\tau_1;\tau_2]\ e$
	$\mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e)$	$\mathtt{inr}[\tau_1;\tau_2]\ e$
	$\mathtt{case}(e; x_1.e_1; x_2.e_2)$	$\mathtt{case}\ e\ \{\mathtt{inl}\ x_1 \Rightarrow e_1 \mid \mathtt{inr}\ x_2 \Rightarrow e_2\}$

Figure 3: PCF language reference

we have implemented `let` bindings, `lambda` abstractions and `case` matching. (If zero statement, i.e. `ifz`, remains to be a sub part of `case` matching.)

3.2 Representations

At this part, it is essential to discuss how we introduce and eliminate these operations on the primitives. Appendix contains all the introduction and eliminations rules.

A very short overview of this process can be done through considering a single connective. For the sake of an example, we are going to consider `pair`.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}(\mathtt{pair}\text{-I})$$

$$\left[\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{l}](e) : \tau_1}(\mathtt{pair}\text{-E}_1)\right] \qquad \left[\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{r}](e) : \tau_2}(\mathtt{pair}\text{-E}_2)\right]$$

Basically, given two expressions of type τ_1 and τ_2, we can use `pair-I` which is the pair introduction rule to create a pair of type $\tau_1 \times \tau_2$.

Now that we have a pair, we would like to eliminate it in order to get back the left or the right element of the pair. We can do this by eliminating the `pair` constructor with the `projection` constructor. There are two elimination rules for pair and they evaluate to the left or the right element of the pair. These rules are `pair`-E$_1$ and `pair`-E$_2$ in respective order. They project the left or right element depending on which rule is chosen.

Rest of the introduction and elimination rules are similar and can easily be understood by looking at the appendix.

3.3 Translation

Now that the typing rules are introduced, we can proceed to the actual translation of the PCF language to `C0` bytecode. First and foremost, we need to learn about the `C0` bytecode. A strict subset of the `C0` bytecode instruction set is given in the appendix of this paper.

The second step would be to learn and understand the translation process. For this, for the sake of consistency, we will again be talking about the connective `pair` as well as some other interesting aspects of other connectives.

The following bytecode sequence is used to create any expression given in the form $\langle e_1, e_2 \rangle : \tau_1 \times \tau_2$.

```
 BB 10       # new 16
 18 59       # dup
<bytecode instructions for e1>
 20 4F       # amstore
 21 59       # dup
 22 62 08    # aaddf 8
<bytecode instructions for e2>
 26 4F       # amstore
 27 B0       # return
```

Basically, what we are doing in this case is we are first allocating an array of size 16 which is enough space to hold two 8 byte pointers. Then we are storing the pointer to $e_1 : \tau_1$ as the first 8 bytes of the array as well as pointer to $e_2 : \tau_2$ as the second 8 bytes of the array. We are doing this process without using any local variables, hence arises the need to duplicate the pointer to the head of the pair structure, in this case it is a 16 byte array.

If we wanted to eliminate the pair, we would need to do so by eliminating it with a projection constructor. Turns out eliminating a pair is significantly simpler than introducing one. If we wanted to get the left projection $e_1 : \tau_1$ of the pair $\langle e_1, e_2 \rangle : \tau_1 \times \tau_2$, we simply follow the following bytecode instructions.

```
<bytecode for creating the pair>
2F       # amload
```

Here what we are doing is just accessing the first 8 bytes of the array since we are looking for the left projection.

Similarly, if we wanted to get the right projection $e_e : \tau_e$ of the pair $\langle e_1, e_2 \rangle : \tau_1 \times \tau_2$, then we do:

```
<bytecode for creating the pair>
 62 08    # aaddf 8
 28 2F    # amload
```

Since this time we are looking for the right projection of the array, we first need to move our pointer by 8 bytes and then access the first 8 bytes of the array.

In addition to `pair`, another interesting connective that this paper wants to touch upon is `let` bindings because `let` bindings use an interesting conversion process to `lambda` expressions as well as they make use of local variables.

STUFF ABOUT LET

4. INTERESTING DECISIONS OF THE COMPILATION

In this section, we are going talk about interesting decision we as the authors of this paper had to make while creating this project.

4.1 Meta Bytecode Instructions

4.2 Type Checking in Bytecode Instruction Level

5. CONCLUSIONS

6. ACKNOWLEDGMENTS

7. APPENDIX

7.1 PCF

$$\frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau}(\text{var}) \qquad \frac{}{\Gamma\vdash \mathtt{z}:\mathtt{nat}}(\mathtt{nat}\text{-I}_1)$$

$$\frac{\Gamma\vdash e:\mathtt{nat}}{\Gamma\vdash \mathtt{s}(e):\mathtt{nat}}(\mathtt{nat}\text{-I}_2)$$

$$\left[\frac{\Gamma\vdash e:\mathtt{nat} \qquad \Gamma\vdash e_0:\tau \qquad \Gamma, x:\mathtt{nat}\vdash e_1:\tau}{\Gamma\vdash \mathtt{ifz}(e;e_0;x.e_1):\tau}(\mathtt{nat}\text{-E})\right]$$

$$\frac{\Gamma\vdash e_1:\tau_1 \qquad \Gamma, x:\tau_1\vdash e_2:\tau}{\Gamma\vdash \mathtt{let}(e_1;x.e_2):\tau}(\text{let})$$

$$\frac{\Gamma, x:\tau_1\vdash e:\tau_2}{\Gamma\vdash \mathtt{lam}[\tau_1](x.e):\tau_1\rightharpoonup\tau_2}(\rightharpoonup\text{-I})$$

$$\frac{\Gamma\vdash e_1:\tau_1\rightharpoonup\tau_2 \qquad \Gamma\vdash e_2:\tau_1}{\Gamma\vdash \mathtt{ap}(e_1,e_2):\tau_2}(\rightharpoonup\text{-E})$$

$$\frac{\Gamma, x:\tau\vdash e:\tau}{\Gamma\vdash \mathtt{fix}[\tau](x.e):\tau}(\text{fix})$$

7.2 Products

$$\frac{}{\Gamma\vdash\langle\rangle:\mathtt{unit}}(\mathtt{unit}\text{-I}) \qquad \frac{\Gamma\vdash e_1:\tau_1 \qquad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash\langle e_1,e_2\rangle:\tau_1\times\tau_2}(\times\text{-I})$$

$$\left[\frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{pr}[\mathtt{l}](e):\tau_1}(\times\text{-E}_1)\right] \qquad \left[\frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{pr}[\mathtt{r}](e):\tau_2}(\times\text{-E}_2)\right]$$

7.3 Sums

$$\frac{\Gamma\vdash e:\mathtt{void}}{\Gamma\vdash \mathtt{abort}[\tau](e):\tau}(\mathtt{void}\text{-E})$$

$$\frac{\Gamma\vdash e:\tau_1}{\Gamma\vdash \mathtt{in}[\tau_1;\tau_2][\mathtt{l}](e):\tau_1+\tau_2}(+\text{-I}_1)$$

$$\frac{\Gamma\vdash e:\tau_2}{\Gamma\vdash \mathtt{in}[\tau_1;\tau_2][\mathtt{r}](e):\tau_1+\tau_2}(+\text{-I}_2)$$

$$\left[\frac{\Gamma\vdash e:\tau_1+\tau_2 \qquad \Gamma, x_1{:}\tau_1\vdash e_1:\tau \qquad \Gamma, x_2{:}\tau_2\vdash e_2:\tau}{\Gamma\vdash \mathtt{case}(e;x_1.e_1;x_2.e_2):\tau}(+\text{-E})\right]$$

7.4 Subset of *C0* Bytecode Instructions

```
Stack operations

0x59 dup            S, v -> S, v, v
0x57 pop            S, v -> S
0x5F swap           S, v1, v2 -> S, v2, v1
```

```
Arithmetic

0x60 iadd         S, x:w32, y:w32 -> S, x+y:w32
0x6C idiv         S, x:w32, y:w32 -> S, x/y:w32
0x68 imul         S, x:w32, y:w32 -> S, x*y:w32
0x64 isub         S, x:w32, y:w32 -> S, x-y:w32

Local Variables

0x15 vload <i>      S -> S, v         v = V[i]
0x36 vstore <i>     S, v -> S         V[i] = v

Constants

0x10 bipush <b>    S -> S, x:w32

Control Flow

0x00 nop                S -> S
0x9F if_cmpeq <o1,o2>  S, v1, v2 -> S
0xA0 if_cmpne <o1,o2>  S, v1, v2 -> S
0xA1 if_icmplt <o1,o2> S, x:w32, y:w32 -> S
0xA2 if_icmpge <o1,o2> S, x:w32, y:w32 -> S
0xA3 if_icmpgt <o1,o2> S, x:w32, y:w32 -> S
0xA4 if_icmple <o1,o2> S, x:w32, y:w32 -> S
0xA7 goto <o1,o2>      S -> S

Memory

0xBB new <s>       S -> S, a:*

0x62 aaddf <f>      S, a:* -> S, (a+f):*

0x2F amload    S, a:* -> S, b:*     (b = *a)
0x4F amstore   S, a:*, b:* -> S     (*a = b)
```