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	$\|\|$	$\texttt{+}, Q$	$\longrightarrow$	$S, x + y$	$\|\|$	Q
S, x, y	$\|\|$	$\texttt{-}, Q$	$\longrightarrow$	$S, x - y$	$\|\|$	Q
S, x, y	$\|\|$	$\texttt{*}, Q$	$\longrightarrow$	$S, x * y$	$\|\|$	Q
S, x, y	$\|\|$	$\texttt{/}, Q$	$\longrightarrow$	$S, x \,/\, y$	$\|\|$	Q
S, x, y	$\|\|$	$\texttt{Pair}, Q$	$\longrightarrow$	$S, \langle x, y\rangle$	$\|\|$	Q
$S, \langle x, y\rangle$	$\|\|$	$\texttt{Prj1}, Q$	$\longrightarrow$	S, x	$\|\|$	Q
$S, \langle x, y\rangle$	$\|\|$	$\texttt{Prj2}, Q$	$\longrightarrow$	S, y	$\|\|$	Q
$S, 0$	$\|\|$	$\texttt{If}, tok_1, tok_2, Q$	$\longrightarrow$	S, tok_1	$\|\|$	Q
$S, 1$	$\|\|$	$\texttt{If}, tok_1, tok_2, Q$	$\longrightarrow$	S, tok_2	$\|\|$	Q
S, n	$\|\|$	$\texttt{Skip}, Q$	$\longrightarrow$	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
3.1 Language Constructors
3.2 Representations
3.3 Translation
4. CONCLUSIONS
5. ACKNOWLEDGMENTS
6. APPENDIX
You can compile your files using `CM.make "sources.cm"`. We have provided three (!) ways to test the final implementation: an interpreter, a test harness and a reference

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

implementation. All of these are based on a parser we provide for you (see examples below).

6.1 Interpreter

As usual, to run the interpreter, execute `TopLevel.repl();`. This will provide a command-line interpreter that will provide two basic commands, `step` and `eval`. These commands do not take a mode argument any more as we have only one kind of dynamics.

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. This is the second column in the table in which introduce the syntax for a language. We provide below the grammar that the interpreter accepts, as well as a sample session of the interpreter.

pcf-grammar.txt interpreter-session.txt

6.2 Test Harness

Another way to test your code is by `TestHarness.runalltests(v);` where `v` is a `bool` indicating whether you want verbose output or not. This is mostly just a framework set up for you, in `tests.sml`, with a few simple test cases. You are responsible for handing in a working solution. Although not sufficient, this means handing in a well-tested implementation. You need to come up with test cases to exercise your code. In order to generate a comprehensive suite of tests, you are encouraged to share test cases with your classmates.

6.3 Reference Implementation

Finally, we have included the solution to both sections of this assignment as a binary heap image, `ref_impl`. You can load it into SML by passing in the `@SMLload=ref_impl` flag. Your solution should behave just like ours (if you find a bug in our implementation, extra credit to you!)

7. STATICS

Rules for explicit eliminatory forms are bracketed as explained in Section 3.

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

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}}(\mathtt{unit}\text{-I}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}(\times\text{-I}) \qquad \left[\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{l}](e) : \tau_1}(\times\right.$$

7.3 Sums

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

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

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

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

8. DYNAMICS (EAGER, LEFT-TO-RIGHT)

Rules for explicit eliminatory forms are bracketed as explained in Section 3.

8.1 PCF with Natural Numbers and Let

$$\frac{}{\texttt{z}\ \mathsf{val}}(\mathrm{nat}^1_{\mathsf{v}}) \qquad \frac{e\ \mathsf{val}}{\texttt{s}(e)\ \mathsf{val}}(\mathrm{nat}^2_{\mathsf{v}}) \qquad \frac{}{\texttt{lam}[\tau](x.e)\ \mathsf{val}}(\rightharpoonup_{\mathsf{v}})$$

$$\frac{e \mapsto e'}{\texttt{s}(e) \mapsto \texttt{s}(e')}(\mathrm{s}_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\texttt{ifz}(e; e_0; x.e_1) \mapsto \texttt{ifz}(e'; e_0; x.e_1)}(\mathrm{ifz}_{\mathsf{s}})\right] \quad \left[\frac{}{\texttt{ifz}(\texttt{z}; e_0; x.e_1) \mapsto e_0}(\mathrm{ifz}^1_{\mathsf{e}})\right] \quad \left[\frac{\texttt{s}(e)\ \mathsf{val}}{\texttt{ifz}(\texttt{s}(e); e_0; x.e_1) \mapsto [e/x]e_1}(\mathrm{ifz}^2_{\mathsf{e}})\right]$$

$$\frac{e_1 \mapsto e_1'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1'; e_2)}(\mathrm{ap}^1_{\mathsf{s}}) \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\texttt{ap}(e_1; e_2) \mapsto \texttt{ap}(e_1; e_2')}(\mathrm{ap}^2_{\mathsf{s}})$$

$$\frac{e_2\ \mathsf{val}}{\texttt{ap}(\texttt{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}(\mathrm{ap}_{\mathsf{e}})$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1; x.e_2) \mapsto \texttt{let}(e_1'; x.e_2)}(\mathrm{let}_{\mathsf{s}}) \quad \frac{e_1\ \mathsf{val}}{\texttt{let}(e_1; x.e_2) \mapsto [e_1/x]e_2}(\mathrm{let}_{\mathsf{e}}) \quad \frac{}{\texttt{fix}[\tau](x.e) \mapsto [\texttt{fix}[\tau](x.e)/x]e}(\mathrm{fix}_{\mathsf{s}})$$

8.2 Products

$$\frac{}{\langle\rangle\ \mathsf{val}}(\texttt{unit}_{\mathsf{v}}) \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}}(\times_{\mathsf{v}})$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}(\times^1_{\mathsf{s}}) \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2\rangle \mapsto \langle e_1, e_2'\rangle}(\times^2_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\texttt{pr}[\texttt{l}](e) \mapsto \texttt{pr}[\texttt{l}](e')}(\mathrm{prl}_{\mathsf{s}})\right] \quad \left[\frac{e \mapsto e'}{\texttt{pr}[\texttt{r}](e) \mapsto \texttt{pr}[\texttt{r}](e')}(\mathrm{prr}_{\mathsf{s}})\right] \quad \left[\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{pr}[\texttt{l}](\langle e_1, e_2\rangle) \mapsto e_1}(\mathrm{prl}_{\mathsf{e}})\right] \quad \left[\frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\texttt{pr}[\texttt{r}](\langle e_1, e_2\rangle) \mapsto e_2}(\mathrm{prr}_{\mathsf{e}})\right]$$

8.3 Sums

$$\frac{e \mapsto e'}{\texttt{abort}[\tau](e) \mapsto \texttt{abort}[\tau](e')}(\mathrm{abort}_{\mathsf{s}}) \qquad \frac{e\ \mathsf{val}}{\texttt{in}[\tau_1; \tau_2][\texttt{l}](e)\ \mathsf{val}}(+^1_{\mathsf{v}})$$

$$\frac{e\ \mathsf{val}}{\texttt{in}[\tau_1; \tau_2][\texttt{r}](e)\ \mathsf{val}}(+^2_{\mathsf{v}})$$

$$\frac{e \mapsto e'}{\texttt{in}[\tau_1; \tau_2][\texttt{l}](e) \mapsto \texttt{in}[\tau_1; \tau_2][\texttt{l}](e')}(+^1_{\mathsf{s}})$$

$$\frac{e \mapsto e'}{\texttt{in}[\tau_1; \tau_2][\texttt{r}](e) \mapsto \texttt{in}[\tau_1; \tau_2][\texttt{r}](e')}(+^2_{\mathsf{s}})$$

$$\left[\frac{e \mapsto e'}{\texttt{case}(e; x_1.e_1; x_2.e_2) \mapsto \texttt{case}(e'; x_1.e_1; x_2.e_2)}(\mathrm{case}_{\mathsf{s}})\right]$$

$$\left[\frac{e\ \mathsf{val}}{\texttt{case}(\texttt{in}[\tau_1; \tau_2][\texttt{l}](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1}(\mathrm{case}^1_{\mathsf{e}})\right]$$

$$\left[\frac{e\ \mathsf{val}}{\texttt{case}(\texttt{in}[\tau_1; \tau_2][\texttt{r}](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2}(\mathrm{case}^2_{\mathsf{e}})\right]$$