CMSC 22610 **Computer Language Implementation** **Handout 5**
Winter 2005 **March 3**

MLR code generation

1 Introduction

This handout describes the code generation API that you will use for Project 4 and gives some hints about code generation for MLR features.

2 The code generation API

The code generation API is organized into three modules. The `Emit` module implements code streams, which are an abstraction of the generated output file; the `Labels` module implements labels for naming code locations, and the `Instructions` module implements an abstract type of VM instructions. Each of these modules is described below.

2.1 Code streams

A code stream provides a container to collect the instructions emitted by your code generator. You create a code Once code generation is complete, you invoke the `finish` operation which does an assembly pass and then writes the binary object file to disk. The `Emit` module also provides hooks for registering string literals and C functions.

2.2 Labels

The `Labels` module defines an abstract type of label that is used to represent code locations. The `Emit` structure provides the `defineLabel` function for associating a label with the current position in the code stream, and the control-flow instructions take labels as arguments. There is also an instruction for pushing the value of a label on the stack, which is required to create closures (see Section 4).

2.3 Instructions

The `Instructions` module provides an abstract type that represents VM instructions. For those instructions that take arguments, it provides constructor functions and for those without arguments, it provides abstract values.

3 Implementing records

Because MLR supports record polymorphism, your code generator must support a mechanism for mapping labels to slots. There are a number of different techniques for this problem, we will use one owed to Didier Rémy, which exploits the fact that we have the whole program available to the compiler. For each record label that appears in the program, assign a unique integer ID id_l. Then, for any record type $\mathbf{\{} l : \tau_l{}^{l \in \mathcal{L}} \mathbf{\}}$, find the least $p_{\mathcal{L}}$, such that for any $l_1, l_2 \in \mathcal{L}$,

$$\mathrm{id}_{l_1} \bmod p_{\mathcal{L}} = \mathrm{id}_{l_2} \bmod p_{\mathcal{L}} \iff l_1 = l_2$$

Assume that the labels of $\mathcal{L}$ are sorted and that $\mathrm{index}_{\mathcal{L}}(l)$ is the index of the label l in the sorted order (counting from 1). We can construct an array $H_{\mathcal{L}}$ of size $p_{\mathcal{L}} + 1$, such that $H_{\mathcal{L}}[0] = p_{\mathcal{L}}$ and $H_{\mathcal{L}}[(\mathrm{id}_l \bmod p_{\mathcal{L}}) + 1] = \mathrm{index}_{\mathcal{L}}(l)$. Then the representation of a record of type $\mathbf{\{} l : \tau_l{}^{l \in \mathcal{L}} \mathbf{\}}$ is an $n{+}1$ element array r, where $r[0] = H_{\mathcal{L}}$ and $r[\mathrm{index}_{\mathcal{L}}(l)]$ holds the value of the field labeled by l. Note that the $H_{\mathcal{L}}$ record is independent of the types of the fields; it only depends on the set of labels.

Using this representation, the VM code for the expression “`a.x`” (assuming that `a` is a local variable) is

```
loadlocal(a)
push(0)
select(0)
int(id_x)
push(1)
select(0)
mod
index
index
```

For example, assume that we have a program with labels `x`, `y`, and `z`, and that $\mathrm{id}_{\mathtt{x}} = 840$, $\mathrm{id}_{\mathtt{y}} = 841$, $\mathrm{id}_{\mathtt{z}} = 842$. For a record type with labels $\mathcal{L} = \{\mathtt{x}, \mathtt{z}\}$, $p_{\mathcal{L}} = 3$ and $H_{\mathcal{L}} = [3, \mathrm{index}_{\mathcal{L}}(\mathtt{x}), -, \mathrm{index}_{\mathcal{L}}(\mathtt{z})]$, where the value in $H_{\mathcal{L}}[2]$ is not associated with a label and does not matter.

To support this implementation strategy for records, you will have to analyze the program to determine the set of labels and the set of distinct record types that are constructed (record types that appear in type constraints do not affect the representation).

4 Implementing function closures

Function closures are used to represent MLR function values. Closures are represented by heap-allocated arrays, where the first element holds the address of the function’s code and the remaining slots hold the function’s free variables. For example, consider the function

```
fun add (x : int) (y : int) : int = (x + y)
```

The `add` function returns a function when given one argument, so we can think of it as being equivalent to:

```
fun add (x : int) : int -> int =
  fun add$inner (y : int) : int = (x + y) in add$inner
```

Thus, the code for `add` might look like the following:

Label	Instruction	Comment
`add:`	**`entry`**`(0)`	
	`label``(add$inner)`	
	`loadlocal``(2)`	push `x`
	`alloc``(2)`	allocate closure
	`storelocal``(2)`	store result
	`ret`	
`add$inner:`	**`entry`**`(0)`	
	`loadglobal`(1)	push global variable `x`
	`loadlocal``(2)`	push `y`
	`add`	
	`storelocal``(2)`	store result
	`ret`	

In addition to the free variables of a function, a function closure should also include any record header values used to construct records in the function.

5 Bootstrapping

The VM starts execution with the top of the stack pointing to the command-line arguments and the PC pointing to the first instruction in the code stream, but it is unlikely that the compiled `main` function will be at that location. Furthermore, you must initialize the pervasive environment (*e.g.*, the `IO` record) and create the closures for the top-level functions. Thus, you should view a program

```
fun f1 (x : ty) : ty′ = ...
...
fun main (x : string list) : int = ...
```

as being in a context roughly like

```
let IO = { ... } in
let String = { ... } in
let Int = { ... } in
...
fun f1 (x : ty) : ty′ = ...
...
fun main (x : string list) : int = ... in
  main (args)
```

The initialization code should also include construction of record headers and the instruction following the call to `main` should be a **`halt`** instruction.

6 Instruction encodings

Most instructions in the VM are either one, two, or three bytes long.[1] The first byte is consists of a two-bit length field (bits 6 and 7), and a six-bit opcode field (bits 0-5). The length field encodes

[1]The one exception if the **`int`** instruction, which has a five byte form.

the number of extra instruction bytes (*i.e.*, zero for one-byte instructions, one for two-byte instructions, and two for three-byte instructions). In the case of the two and three byte instructions, the extra bytes contain immediate data (*e.g.*, the offset of a `load` instruction), which is stored in 2's complement big-endian format.[2] Figure 1 gives a list of the instructions and their lengths; note that some instructions have both one and two or two and three-byte forms. The actual opcodes for the VM instructions are given in the `opcode.sml` file, which is part of the sample code.

7 Document history

Mar. 3 Original version.

Mar. 12 Fixed code fragment for record indexing (Section 3).

[2]The term "big-endian" means that the most significant byte comes first. For example, the number 513 is represented as the byte sequence $2, 1$.

Instruction	**Length**	**Comment**
`add`, **`sub`**, **`mul`**, **`div`**, **`mod`**, **`neq`**, **`equ`**, **`less`**, **`lesseq`**, **`not`**	1	
`alloc`(n)	2	if $0 \leq n < 256$
`alloc`(n)	3	if $256 \leq n < 2^16$
`select`(i)	2	if $0 \leq i < 256$
`select`(i)	3	if $256 \leq i < 2^16$
`update`, **`index`**	1	
`int`(n)	2	if $-128 \leq n < 128$
`int`(n)	3	if $n < -128$ or $128 \leq n$
`int`(n)	5	if $n < -2^15$ or $2^15 \leq n$
`literal`(n)	2	if $-128 \leq n < 128$
`literal`(n)	3	if $n < -128$ or $128 \leq n$
`label`(n)	2	if $-128 \leq n < 128$
`label`(n)	3	if $n < -128$ or $128 \leq n$
`swap`	1	
`swap`(n)	2	$0 \leq n < 256$
`push`(n)	2	$0 \leq n < 256$
`pop`	1	
`pop`(n)	2	$0 \leq n < 256$
`loadlocal`(n)	2	$-128 \leq n < 128$
`loadlocal`(n)	3	if $n < -128$ or $128 \leq n$
`storelocal`(n)	2	$-128 \leq n < 128$
`storelocal`(n)	3	if $n < -128$ or $128 \leq n$
`loadglobal`(n)	2	$n < 256$
`loadglobal`(n)	3	if $256 \leq n < 2^{16}$
`pushep`, **`popep`**	1	
`jmp`(n)	2	if $-128 \leq n < 128$
`jmp`(n)	3	if $n < -128$ or $128 \leq n$
`jmpif`(n)	2	if $-128 \leq n < 128$
`jmpif`(n)	3	if $n < -128$ or $128 \leq n$
`call`(n)	2	if $-128 \leq n < 128$
`call`(n)	3	if $n < -128$ or $128 \leq n$
`entry`(n)	2	$0 \leq n < 256$
`entry`(n)	3	if $256 \leq n < 2^{16}$
`ret`, **`tailcall`**	1	
`ccall`(i)	2	$0 \leq i < 256$
`nop`, **`halt`**	1	

Figure 1: VM instruction lengths