CMSC 22620 Spring 2009 | Implementation of Computer Languages | Project 2 April 14, 2009

Simple code generation

Due: May 1, 2009

1 Introduction

In this project, you will be implementing a simple assembly-code generator for LangF. This phase of the compiler will take as input a first-order graph-based representation of the program and will produce x86-64 assembly code. A companion handout describes the x86-64 instructions that you will use.

2 Runtime representations

All LangF values are represented by a single machine word. In many cases, this word is a pointer to heap-allocated storage, but it might also be an immediate value. We refer to values that are represented as pointers as *boxed* values, while values that are represented as immediate integers are *unboxed*. Because the garbage collector might confuse an immediate value with a pointer, we use a tagged representation for immediate integers. The integer value n is represented as the tagged integer value $2n+1$. Because tagged values are always odd, they will not be confused with pointers.

The code that you generate will have to deal with four different kinds of values that can be stored in 64-bit registers. These are pointers to heap objects, code addresses, tagged integers, and raw integers. The following type system defines the relationship between these types, where T is the type of any value that can appear in a register, U is the type of uniformly-represented values that represent LangF types, and M is the type of heap objects.

$$
\begin{array}{rcl}
T & ::= & U \\
 & | & \mathbf{int64} \\
 & | & \mathbf{codeaddr} \\
U & ::= & \mathbf{taggedint} \\
 & | & \mathbf{ptr}(M) \\
M & ::= & \mathbf{tuple}(T_1, \ldots, T_n) \\
 & & \mathbf{array}(U_1, \ldots, U_n) \\
 & & \mathbf{string}
\end{array}
$$

We say that a LangF type τ has a *boxed* representation if it is always represented by a pointer value and an *unboxed* representation if it is always represented as a tagged integer. If its repre-

sentation can be either (as is the case for some datatypes), then we say that its representation is *mixed*.

There are three different kinds of heap objects used by the LangF system:

1. Tuples, which are sequences of one or more values, including raw integers and code addresses.
2. Arrays, which are mutable sequences of zero or more uniform values.
3. Strings, which are sequences of zero or more bytes. Strings are padded to a multiple of 8 bytes.

Every heap object has a header word. The low 2 bits of the word determine the of heap object and the other 62 bits store other information as described in the following table:

$b_{63} \cdots b_8$	$b_7 \cdots b_2$	b_1	b_0	Object type
forwarded-object address		0	0	GC forward pointer
pointer mask	length	0	1	Tuple
length		1	0	Array
length		1	1	String

The length fields represent the length in elements (*i.e.*, bytes for strings and quadwords for tuples and arrays). The tuple-object header also includes a 56-bit mask that tells the garbage collector which fields contain non-uniform values (*i.e.*, raw 64-bit integers). A bit in this mask is set to 1 if the corresponding field of the tuple is non-uniform. For example, the LangF `Integer` type is represented as a one non-uniform word tuple. Its header is

`00 ... 01`	`000001`	`01`

Tuples with more than 56 elements are not supported.

2.1 Representation of primitive types

The primitive LangF types have boxed representation as follows:

LangF type	Representation
`Integer`	$\mathbf{ptr}(\mathbf{int64})$
`String`	$\mathbf{ptr}(\mathbf{string})$
`Array[]`	$\mathbf{ptr}(\mathbf{array}(-))$

Note that the `Integer` type is represented by a pointer to a heap-allocated object containing the 8-byte integer value. To compute with these values requires fetching arguments from memory and then storing the results in freshly allocated objects.

2.2 Representation of functions

A function is represented as a heap-allocated pair of a code address and a pointer to the function's environment tuple.

$$\mathbf{ptr}(\mathbf{tuple}(\mathbf{codeaddr}, \mathbf{ptr}(\mathbf{tuple}(\cdots))))$$

Mutually recursive functions share the same environment tuple.

2.3 Representation of datatypes

The other LangF types are generated by datatype declarations. Their representation depends on the particular mix of constructors that they have. Let t be a datatype with m nullary constructors ($C_1, \ldots, C_m$) and k data-constructor functions $F_1\{\vec{\tau}_1\}, \ldots, F_k\{\vec{\tau}_k\}$. We say that $\vec{\tau}$ is *unboxed* if $|\vec{\tau}| = 1$ and τ_1 is unboxed; we say that $\vec{\tau}$ is *mixed* if $|\vec{\tau}| = 1$ and τ_1 is mixed; otherwise we say that $\vec{\tau}$ is boxed. Also let $n_i = |\vec{\tau}_i|$ be the arity of F_i. Then the following table gives the representation of the various constructors based on the number of constructors and the representations of the τ's.

m	k	n_1	C_i	$F_j(\vec{v})$	t's representation
> 0	0	n.a.	i	n.a.	unboxed
0	1	1	n.a.	$\vec{v}$	$\vec{\tau}_1$'s representation
0	1	> 1	n.a.	$\mathbf{tuple}(\vec{v})$	boxed
> 0	1	1	i	$\vec{v}$ if $\vec{\tau}_1$ is boxed.	mixed
> 0	1	1	i	$\mathbf{tuple}(\vec{v})$ if $\vec{\tau}_1$ is unboxed or mixed.	mixed
> 0	1	> 1	i	$\mathbf{tuple}(\vec{v})$	mixed
≥ 0	> 1	n.a.	i	$\mathbf{tuple}(j, v)$	mixed

In this chart, i means the immediate value i and $\mathbf{tuple}(\cdots)$ means a heap-allocated tuple. Applying this algorithm to the the builtin datatypes we get:

$$\begin{aligned} \texttt{Unit} &\rightarrow 0 \\ \texttt{False} &\rightarrow 0 \\ \texttt{True} &\rightarrow 1 \\ \texttt{None} &\rightarrow 0 \\ \texttt{Some}[\tau](a) &\rightarrow \mathbf{tuple}(a) \end{aligned}$$

Note that while for boxed values of τ, we could represent $\texttt{Some}[\tau](a)$ directly as a, this representation would require using runtime type information, which we do not have.

3 Register conventions

A small number of the x86-64's 16 registers are reserved for special use:

Register	Purpose
`%rsp`	Stack pointer
`%rbp`	Frame pointer
`%rbx`	Allocation pointer
`%r12`	Heap-limit pointer
`%rdi`	Argument register for standard calls
`%rsi`	Handler pointer for standard calls
`%rdx`	Environment pointer for standard calls
`%rax`	Result register for standard calls

The other registers are available for use as miscellaneous registers, including to pass arguments to known functions, but note that `%r13`-`%r15` are callee-save registers in the C ABI.

4 Calling conventions

Since function calls are an important part of functional languages, we use a variety of different calling conventions depending on the circumstance.

4.1 Standard calls

For standard calls, we follow the basic calling conventions defined by the ABI. A standard call will involve three arguments:

1. The argument value in register `%rdi`.
2. The handler pointer in register `%rsi`.
3. The environment pointer in register `%rdx`.

A standard call returns one result in `%rax`.

4.2 Standard tail calls

A standard tail call is similar to a standard call, except that the caller's stack frame should be deallocated prior to transferring control. For example, assuming that `%rax` holds the code address of the function being called, then we can execute a tail call as follows:

```
leave
jmp   *%rax
```

This code has the effect of discarding the caller's stack and then transferring control to the callee while leaving the original return address on top of the stack.

4.3 Known-function calls

A known-function call is one in which we know all of the call sites of the function and, thus, can specialize the calling convention to the situation. With know functions, we may be able to pass more arguments in registers. While the ABI restricts arguments to six registers, since we are specializing the call, we can use any caller-save register for arguments.[1]

4.4 Known-function tail calls

A known-fucntion call in the tail position can use additional registers for arguments and results, but like the standard-call case, it must pop off the caller's frame before the call.

[1] Note that the ABI permits additional arguments to be passed on the stack, but this complicates tail calls, so we limit arguments to registers.

5 Heap allocation and garbage collection

The `ALLOC` operation in the cluster representation is used to allocate tuple objects (arrays and strings are allocated by runtime-system functions). Since the tuple header requires a bitmask to specify which fields are pointers, we attach a boolean flag to each variable that is true if the variable holds a raw 64-bit integer value.

The cluster representation includes GC-test operations that check to see if we need to perform a garbage collection. To simplify these tests, we set the limit pointer to 8,192 bytes (1024 words) below the actual top of the heap. The GC-test operation has a parameter that specifies the amount of memory required. If this amount is less than 8,192, then we GC-test can be implemented as a simple comparison of the limit pointer (`%r12`) and the allocation pointer (`%rbx`). When a GC test fails, we call the `CallGC` function (provided as part of the runtime system) with a bit mask that specifies the live registers. A subsequent handout will provide more details about the GC protocol.

6 The code generation API

The code generation API is organized into three modules, which live in the `x86-64` directory. The `Emit` module implements code streams, which are an abstraction of the generated output file; the `Label` module implements labels for naming code and data locations, and the `Instruction` module implements an abstract type x86-64 instructions. Each of these modules is described below.

6.1 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.

6.2 Labels

The `Label` 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.

6.3 Instructions

The `Instruction` module provides abstract types that represents x86-64 operands and instructions. For those instructions that have operands, it provides constructor functions and for those without operands, it provides abstract values. Because some immediate values (*e.g.*, the stack frame size) may not be determined at the time that you generate instructions that use them, we provide support for *constant expressions*, which can be used wherever a 32-bit constant is expected. You must define all of the constants the you use before calling the `Emit.finish` function.

7 Hints

This project is not hard, but it does involve significant coding to complete. We recommend that you break the work into two phases. In the first phase, you should focus on getting instruction selection for the cluster representation working without GC support. Once you have basic examples working, you should add the GC support, which will require tracking the liveness of stack locations, etc.