CMSC 22620 Spring 2009	Implementation of Computer Languages	Handout 4 April 28, 2009

Implementing non-local control flow

1 Introduction

The try/escape mechanism in LangF allows non-local control transfers. Implementing such transfers requires some care. In this handout, we describe an approach to implementing this mechanism.

The problem of implementing a non-local control mechanism has two parts: implementing the control transfer and ensuring that live variables are available for access in the handler code. The control transfer is fairly straightforward: it consists of restoring the stack and frame pointers to their values at the point of the handler and then jumping to the handler. Managing the live variables is a bit more difficult. For example, consider the following program:

```
let
  fun f (x : Integer) : Integer = escape[Integer]
  val x : Integer = 1
in
  try f(x) catch x end
end
```

When control returns to the handler, the code needs to be able to access the value of `x`. A smart compiler might put `x` into a callee-save register, which would then have to be restored during the escape. There are two common approaches to ensuring access to the live variables. The first is to force the variables to be saved in the stack across the call.[1] This approach is simple, but can hurt performance in the common case. The alternative is to roll-back the stack frame by frame, restoring callee-save registers as you go. This approach is often used by C++ compilers, which must also invoke the destructors for stack-allocated objects as part of the roll back. For the LangF implementation, we will use the first method.

1.1 Representing handlers

We need three pieces of information for a handler: the code address of the handler, and the stack and frame pointers from when it was created. We can store this information either in the stack or in a heap-allocated tuple.

[1] One can also save them in a heap-allocated object similar to a closure.

1.2 Implementing `try` nd `catch`

The translation phase makes handlers explicit. The above example translates (roughly) into the following LambdaIR code:

```
let f = fn f' (x, h) => escape h
let t = 1
let x = [t]
handler handler066 = return (x)
  apply f05F (x, handler066)
```

In the cluster representation, the handler code is represented as a fragment with parameters that denote the live variables for the fragment. We also have a handler binding form that includes the live variables. The cluster representation of our example is as follows:

```
entry LangFMain$0D7 (_envp0E9) =
  ...
  let t = 1
  let x = [t]
  let handler118 = handler handler117 (x)
  let _cp119 = #0(f)
  let _ep11A = #1(f)
  apply _cp119 (x, handler118, _ep11A)
label handler117 (x') =
  return (x')
  ...
end cluster
```

Note that `x` is live in the handler (where it is named `x'`).

To implement the handler binding, we generate code to allocate and initialize the handler object, which is then bound to `handler118`. When compiling the body of the handler, we will need a mapping from the live variables (*i.e.*, `x'` in the example) to their locations.

1.3 Implementing `escape`

Escape is implemented by loading the stack and frame pointers from the handler and then jumping to the handler's code address.