CMSC 22620/32620 **Implementation of** **Project 3**
Spring 2015 **Computer Languages II** **May 1, 2015**

Closure conversion

Due: May 15, 2015

1 Introduction

In this project, you will be implementing a closure converter for **Flang**. In addition to converting the higher-order LambdaIR representation to a first-order representation, your solution will replace pattern matching with conditionals and lower the representation of data constructors.

2 Closure conversion

The main part of the project is a translation from the LambdaIR, which supports lexically nested and higher-order functions, to the *Cluster* IR in which all functions are first-order and defined at top-level. To implement this conversion, you will first need to perform free-variable analysis on the LambdaIR representation and then need to use the information from that analysis to drive the conversion to clusters.

3 Pattern matching

The Cluster IR does not have data constructors or pattern matching, so one of the tasks in translating a program will be translating the LambdaIR match-case and pattern constructs into Cluster IR code. A LambdaIR match case "`CASE`(*x rules*, *dflt*)" is translated to Cluster IR code that tests x against each pattern until a match is found. If the patterns do not cover all possible values, then a default expression is included. For a rule of the form `{`*pat* `=>` *exp*`}`, we generate code that is based on the syntax of the pattern pat. There are basically two kinds of patterns found in the LambdaIR:[1]

1. A nullary-data-constructor pattern: `{`C `=>` *exp*`}`. If this pattern is the last pattern in the match case, then it is exhaustive (recall that the type checker checks the exhaustiveness of match cases). In that situation, the resulting code is the translation of the right-hand-side expression. If the rule is not the last rule, then we need to test x against the representation of the constructor C.

2. A data-constructor pattern: `{`$C(y)$ `=>` *exp*`}`. If this pattern is the last pattern in the match case, then it is exhaustive (recall that the type checker checks the exhaustiveness of match cases). In that case, the Cluster IR code extract the constructor's argument from the value and bind

[1] Variable and tuple patterns will be covered by the default case.

it to y in the translation of exp. If the rule is not the last rule, then we need to test to see if the argument matches the constructor and, if so, extract the constructor's argument from the value and bind it to y in the translation of exp.

When the type of the match-case argument has mixed representation, we need to test if the representation is boxed or unboxed before checking constructor tag values. We use the primitive operator **isBoxed**, which returns true for heap-allocated values (*i.e.*, tuples and strings) and false for immediate values, to implement this test.

4 Runtime representations

All Flang values are represented by a single 64-bit 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.[2]

4.1 Representation of functions

Flang supports higher-order functions, which requires representing functions as heap-allocated values. For example, consider the function

```
fun add (x : Integer) -> Integer = {
    fun f y = (x + y);
    f
  }
```

When applied to an integer argument n, The `add` function returns a function that will add the integer n to its argument. The representation of the result of `add` must include the value of n. In general, the representation of a function will include the free variables of the function stored in a heap-allocated tuple that we call the function's *environment*. In this example, the environment has a single element (the value of `x`). The representation of a function must also contain the address of the function's code. We could store this address in the environment tuple, but for reasons we explain below, we instead represent a function as a pair of its code address and a pointer to its environment; we call this pair the *closure* of the function.

Things are a bit more complicated with mutually recursive functions. The approach that we take is to share a common environment between the functions, which is why we use the two-level representation of functions. For example, consider

```
fun f (a : Integer) -> Integer = if (a < 0) then 1 else g(a+x)
and g (b : Integer) -> Integer = f(b*y + z)
```

In this case, the environments of `f` and `g` will have three values: `x`, `y`, and `z`. Figure 1 gives a pictorial representation of the representation of `f` and `g`. Note that in this case, since `f` and `g` share the same EP, the calls between the functions can be direct.

[2]Note, however, that the representation of integers is still abstract in the Cluster IR; *i.e.*, the integer `5` is represented as the right-hand-side `NUM(5)`, not `NUM(11)`.

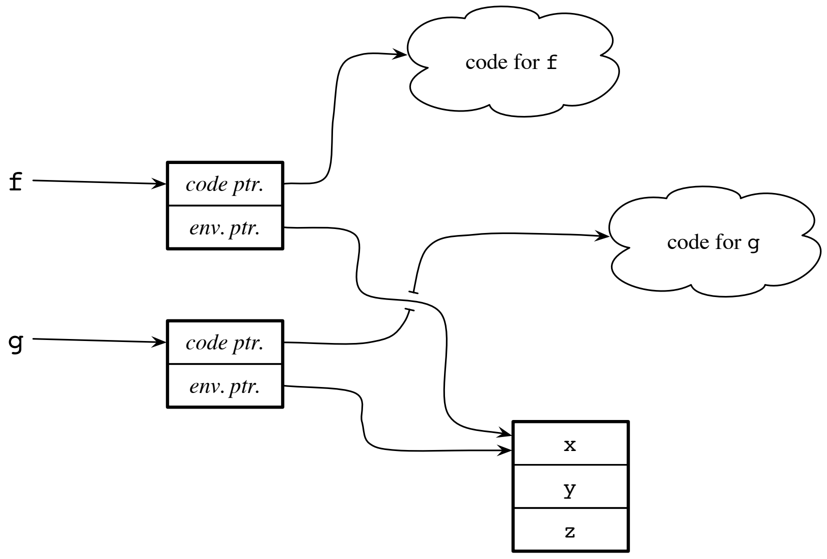

Figure 1: Representation of mutually-recursive functions

4.2 Representation of data constructors

Recall that in Project 2, we determined the representation of data constructors (see Section 2.2 of Project 2). In this project, you will use that representation information to replace nullary data constructors with integer literals (*i.e.*, their tags) and data-constructor functions with the appropriate representation (transparent, boxed, or tagged boxed).

5 Submission

Sources for Project 3 will be seeded into your phoenixforge repositories. These sources include the solution for Project 2, as well as the Cluster representation. You will complete the implementation of the `Closure` module (in `closure/closure.sml`) and commit your solution into the repository. You should not need to change any other file in the source code, but you may wish to add additional files.

We will collect projects from the SVN repositories at 10pm on Wednesday, May 13. Please make sure that you have committed your final version before then.

Revision history

May 1, 2015 Original version