

## Closure Conversion

We have seen that the higher-order nested functions of HOFL and OCAML facilitate abstracting over common processes like the mapping, filtering, accumulation, and generation of data structures (e.g., lists and sets). In languages limited to first-order functions (like FOFL and FOBS), the most straightforward way to express such processes is to specialize these idioms to a particular functional argument — e.g., we can write a special-purpose `map-square` function rather than invoking a generic `map` function with a squaring-function argument. We have also seen that first-class functions enable functional implementations of data structures like pairs, environments, sets, and matrices. It is not at all clear how to express similar implementations in FOFL and FOBS.

In this handout, we explore a technique for expressing typical higher-order function idioms in languages that have only first-order functions. The technique is known as **closure conversion** because within modern compilers it is used to automatically translate the closures of a higher-order functional language into a representation within a first-order functional language. However, it is often helpful to manually perform this technique to simulate higher-order functions in languages that do not fully support them (like C and JAVA()).

### 1 Closure Conversion in FOFL

Let's begin by considering how to simulate higher-order functions in FOFL. We'll start by trying to express in FOFL the HOFL program in Fig. 1. This program includes a function that takes

```
(hofl (n) (list (test 2 3 4 n) (test 5 6 7 n))
  (def (sigma f lo hi)
    (if (> lo hi)
        0
        (+ (f lo) (sigma f (+ lo 1) hi))))
  (def (test p q r h)
    (list (sigma sq 1 h)
          (sigma (scale p) 1 h)
          (sigma (linear q r) 1 h)))
  (def (sq x) (* x x))
  (def (scale c) (fun (y) (* c y)))
  (def (linear a b) (fun (z) (+ (* a z) b))))
```

Figure 1: A HOFL program illustrating higher-order functions.

other functions as arguments (the `sigma` function takes a functional argument `f`) and functions that return functions as results (the `scale` and `linear` functions both return other functions).<sup>1</sup>

Of course, one way to express the program in FOFL is to write three specialized versions of `sigma`, as shown in Fig. 2. However, this is not very satisfying, and it's not the sort of translation that we're looking for. Instead, we would prefer to write a *single* `sigma` function that can apply different functions (squaring, scaling, etc.) for different invocations of `sigma`.

We can do this by developing a simple FOFL representation of the closures that would be created in HOFL. Fig. 3 shows the HOFL closures that would be created as arguments to the `sigma` function

---

<sup>1</sup>Because of HOFL's currying, any function taking more than one argument, such as `sigma`, also illustrates returning functions as results. However, for the present purposes, we will ignore currying unless partially-applied multi-parameter functions are used somewhere in the program. That is, we will pretend as if HOFL supports multi-parameter functions as a kernel construct.

```

(fofl (n) (list (test 2 3 4 n) (test 5 6 7 n))
  (def (sigma-sq lo hi)
    (if (> lo hi)
        0
        (+ (* lo lo) (sigma-sq (+ lo 1) hi))))
  (def (sigma-scale c lo hi)
    (if (> lo hi)
        0
        (+ (* c lo) (sigma-scale c (+ lo 1) hi))))
  (def (sigma-linear a b lo hi)
    (if (> lo hi)
        0
        (+ (+ (* a lo) b) (sigma-linear a b (+ lo 1) hi))))
  (def (test p q r h)
    (list (sigma-sq 1 h)
          (sigma-scale p 1 h)
          (sigma-linear q r 1 h))))

```

Figure 2: A FOFL version of the `sigma` program in that has three specialized versions of `sigma`.

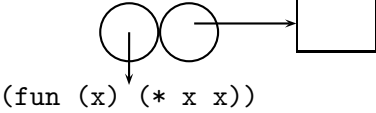
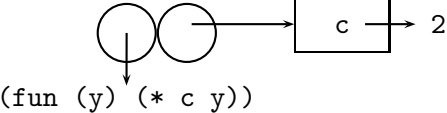
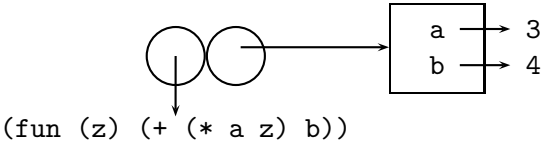
HOFL Closures	FOFL Representations
 $(\text{fun } (x) \text{ } (* x x))$	<code>(list (sym sq))</code>
 $(\text{fun } (y) \text{ } (* c y))$	<code>(list (sym scale) 2)</code>
 $(\text{fun } (z) \text{ } (+ (* a z) b))$	<code>(list (sym linear) 3 4)</code>

Figure 3: Closures from the HOFL example and their representations in FOFL.

in the invocation `(test 2 3 4 10)` and how we can represent these closures in FOFL. Each HOFL closure shows a single environment frame containing just the free variables that appear in the body of the associated function. This is different than the environment chains in the environment-model interpreter, but is all that is really needed to make the closures work as desired.

The FOFL representation that we have chosen for a closure is a list that begins with a symbol that specifies the function part of the closure and whose remaining elements are the values in the closure's environment. The squaring closure has zero environment values, the scaling closure has one (the scaling factor `c`) and the linear function closure has two (the values of `a` and `b`).

Fig. 4 is a FOFL version of the `sigma` program that shows how such closures can be used. The `applyClosure` function is used to apply a one-argument closure to a single argument. It works by performing a dispatch on the symbol in the first slot of the closure list to an appropriate function. The functions `scaleClosed` and `linearClosed` are closed helper functions derived from `scale` and `linear` that pass all needed variables as explicit arguments.

```

(fofl (n) (list (test 2 3 4 n) (test 5 6 7 n))
  (def (sigma f lo hi)
    (if (> lo hi) 0
      (+ (applyClosure f lo) (sigma f (+ lo 1) hi))))
  (def (test p q r h)
    (list (sigma (list (sym sq)) 1 h)
          (sigma (scale 2) p w)
          (sigma (linear q r) 1 h)))
  (def (sq x) (* x x))
  (def (scale c) (list (sym scale) c))
  (def (scaleClosed y c) (* c y))
  (def (linear a b) (list (sym linear) a b))
  (def (linearClosed z a b) (+ (* a z) b))
  (def (applyClosure clo arg)
    (bind name (nth 1 clo) ; Assume nth is 1-based list indexing
      (cond ((sym= name (sym sq)) (sq arg))
            ((sym= name (sym scale))
             (scaleClosed arg (nth 2 clo)))
            ((sym= name (sym linear))
             (linearClosed arg (nth 2 clo) (nth 3 clo)))
            (else (error "unknown closure")))))
  (def (nth i xs) ;; Return value at 1-based index i in list xs
    (cond ((empty? xs) (error "Empty list!"))
          ((= i 1) (head xs))
          (nth (- i 1) (tail xs))))

```

Figure 4: A FOFL program that is the result of closure-converting the `sigma` program from HOFL.

## 2 Closure Conversion in FOFL+

Some languages with only top-level functions (particularly C) allow function values to be named, passed, returned, stored, but they *cannot* be created in any context (i.e., no closures). We can model this functionality in an extension to FOFL we'll call FOFL+. The FOFL+ language is FOFL with two additional expression constructs:

- `(fref  $F$ )` returns the function value denoted by  $F$ . This is like a function pointer value in C.
- `(fapp  $E_{rator} E_{rand_1} \dots E_{rand_n}$ )` invokes the function denoted by  $E_{rator}$  to the values denoted by the operands  $E_{rand_1} \dots E_{rand_n}$ .

Here is a sample FOFL+ program:

```

(fofl-plus (n) (list (app5 (fref inc))
                    (app5 (fref dbl))
                    (app5 (fref mul-n))))
  (def (inc x) (+ x 1))
  (def (dbl y) (* y 2))
  (def (mul-n z) (* z n))
  (def (app5 f) (fapp f 5))

```

Note that in regular FOFL there is no way to write a function like `app5`. However, FOFL+ is still limited compared to HOFL: it is not possible to create nontrivial closures because all functions are created at top-level. The only free variable values that can be closed over are the program parameters (as illustrated by `mul-n` in the above example).

Having function values makes it easier to represent closures in FOFL+. Rather than having the first element of a closure list be a *symbol* standing for a function, we can have it be the appropriate *function value*. Fig. ?? shows how we can express the `sigma` program in FOFL+.

```
(fofl-plus (n) (test n)
  (def (sigma f lo hi)
    (if (> lo hi) 0
        (+ (applyClosure f lo)
            (sigma f (+ lo 1) hi))))
  (def (test w)
    (list (sigma (list (fref sq)) 1 w)
          (sigma (scale 2) 1 w)
          (sigma (linear 3 4) 1 w)))
  (def (sq x clo) (* x x))
  (def (scale c) (list (fref scaleClosed) c))
  (def (scaleClosed y clo)
    (bind c (nth 2 clo)
          (* c y)))
  (def (linear a b) (list (fref linearClosed) a b))
  (def (linearClosed z clo)
    (bindpar ((a (nth 2 clo))
              (b (nth 3 clo)))
              (+ (* a z) b)))
  (def (applyClosure clo arg) (fapp (nth 1 clo) arg clo))
  (def (nth i xs) ;; Return value at 1-based index i in list xs
    (cond ((empty? xs) (error "Empty list!"))
          ((= i 1) (head xs))
          (nth (- i 1) (tail xs)))))
```

Figure 5: A FOFL+ program that is the result of closure-converting the `sigma` program from HOFL.

### 3 Closure Conversion in JAVA

We conclude by considering how to perform closure conversion in JAVA. We will show how to express the `sigma` example in JAVA.

We will use the following interface to represent functions that take a single integer input to an integer result:

```
interface IntFun {public int apply (int x);}

public static int sigma (int lo, int hi, IntFun f) {
  int sum = 0;
  for (int i = lo; i <= hi; i++) {
    sum = sum + f.apply(i);
  }
  return sum;
}

public static void main (String [] args) {
  int n = Integer.parseInt(args[0]);
  System.out.println(sigma(1, n, sqFun()));
  System.out.println(sigma(1, n, scaleFun(2)));
  System.out.println(sigma(1, n, linearFun(3,4)));
}
```

Fig. 6 shows how each of the three functions in the `sigma` program can be represented as a JAVA class.

```
class Sigma {
  :
  public static IntFun sqFun () {return new SqFun ();}
  public static IntFun scaleFun (int c) {return new ScaleFun (c);}
  public static IntFun linearFun (int a, int b)
    {return new LinearFun (a,b);}}

class SqFun implements IntFun {public int apply (int x) {return x * x;}}

class ScaleFun implements IntFun {
  private int c;
  public ScaleFun (int c) {this.c = c;}
  public int apply (int y) {return c * y;}}

class LinearFun implements IntFun {
  private int a,b;
  public LinearFun (int a, int b) {this.a = a; this.b = b;}
  public int apply (int z) {return (a * z) + b;}}
```

Figure 6: A JAVA program that is the result of closure-converting the `sigma` program from HOFL.

Fig. 7 shows how so-called anonymous inner classes can simplify the creation of “functions” in the JAVA program.

```
public static IntFun sqFun () {
  return new IntFun () {public int apply (int x) {return x * x;}};
}

public static IntFun scaleFun (final int c) {
  return new IntFun () {public int apply (int x) {return c * x;}};
}

public static IntFun linearFun (final int a, final int b) {
  return new IntFun () {public int apply (int x) {return (a * x) + b;}};
}
```

Figure 7: Using anonymous inner classes to simplify the expression of functions in JAVA.