Closure Conversion

Di Zhao

zhaodi01@mail.ustc.edu.cn

This is the second assignment of Advanced Topics in Software Technologies. Previously in this course we have learned about the concept of the Continuation-Passing Style (CPS), and implemented the CPS convert function from the ML source language to the continuation-passing language. In this assignment, we will take a step further by looking at another crucial phase in compiling functional languages - closure conversion.

To start with, we will introduce an example to help understanding the concept and purpose of closure conversion. Then you'll need to implement the preparation step for closure conversion - free variables calculating. After that, you need to get familiar with the syntax of the target language in this lab - a closure-passing language. Finally, you will finish the code for closure conversion.

After this assignment, you will gain deeper understanding about functional programming languages.

1 What & Why

In our source language, function definitions have nested static scope, a function can refer to variables of the function in which it is defined. However the notion of a function as a machine-code address does not provide for free variables. To elaborate the problem and its solution, let us consider how to encode this feature in C programming language.

Below is a fragment of code that defines a function `f` inside another function `g`, while `f` refers to the argument `x` of function `g` in its return statement. Namely, there is a free existence of variable `x` in `f`. Without any doubt, C language convention will not allow that. Moreover, to use function `f` later, we want the variable `x` to be accessible even after the function `g` has returned.

```
int (*g (int x))(int) {
  int f (int y){
    return (x+y) ;
  }
  return f;
}
```

Supposing that the above code was legal, to calculate 3 + 4 by using the above function, the main function may look like this:

```
int main (){
  int (*h)(int) = g (3);
  printf ("%d", h (4));
  return 0;
}
```

To solve the problems, we use an *environment* to pass a function the free variables it needs (such as `x` here). And a function will be converted to a *closure* which contains function code and the environment.

Therefore the above code is converted to [1]:

```
void **g (int *env, int x){
  int f (int *env, int y){
    int x = env[0];
    return (x+y) ;
  }
  int *env0 = malloc(sizeof(int)*1);
  env0[0] = x;      //build the environment for f
  void **tuple = malloc(sizeof(void *)*2);
  tuple[0] = f;
  tuple[1] = env0;  //construct the closure
  return tuple;     //return a closure instead of a function
}
```

Functions now take an extra argument `env` as the environment, which contains the original free variables (such as variable `x` in `f` here). `g` will now return a closure instead of a function, the closure contains code for `f` and the environment of `f`. To use the above function to calculate $3 + 4$, we may run:

```
void main (){
  int * env1 = 0;
  void ** closure = g(env1, 3);  //g returns the closure of f.
  void * code = closure[0];   //get function code from the closure
  int env = closure[1];       //get environment from the closure
  printf("%d", ((void * (*)())code)(env, 4));
  //apply the code to the environment and original arguments
}
```

To perform a function call, we need to unfold the corresponding closure and apply the function code (first component of the closure) to the environment (second component of the closure) and its arguments.

To be mentioned, as function `f` can be returned as the result of `g`, or stored into a data structure and invoked after `g` returns, the variables in the activation record of `g` may now be used after `g` has returned. This means that activation

[1] To compile this code with a C compiler, one more step is needed: function `f` has to be lifted to top level. We will look at it in the next lab.

records can no longer be stored on a stack, but must instead be allocated on a heap. We will discuss about this in more detail later in Garbage Collection.

Make sure that you have fully understand the above example before you continue.

2 Calculating free variables

The first step to perform closure conversion is to calculate the free variables inside all function definitions. This allows us to decide which values need to be included in environments (and fetched out from the environments). Only after that will we be able to do closure conversion.

The source language in this lab is exactly the target language in the previous one, i.e., the continuation-passing language. You can review its syntax with Figure 1.

(terms)	K	$\rightarrow$	$\mathsf{letval}\ x\ =\ V\ \mathsf{in}\ K$
		$\mid$	$\mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K$
		$\mid$	$\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ K'$
		$\mid$	$k\ x$
		$\mid$	$f\ k\ x$
		$\mid$	$\mathsf{case}\ x\ \mathsf{of}\ k_1 \parallel k_2$
		$\mid$	$\mathsf{letprim}\ x = PrimOp\ \vec{y}\ \mathtt{in}\ K$
		$\mid$	$\mathsf{if0}\ x\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2$
		$\mid$	$\mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K'$
(values)	V	$\rightarrow$	$()$
		$\mid$	i
		$\mid$	$"s"$
		$\mid$	$(x_1, x_2, \ \ldots\ , x_n)$
		$\mid$	$\mathsf{in}_i\ x$
		$\mid$	$\lambda k\ x.K$
(primitives)	$PrimOp$	$\rightarrow$	$+$
		$\mid$	$-$
		$\mid$	$*$
		$\mid$	print
		$\mid$	$\mathsf{int2string}$

Figure 1: CPS syntax

A free variable of an expression is one that is not bound but is used inside the expression. Such as the y and k in $\mathsf{letval}\ x = y\ \mathsf{in}\ k\ x$. It's worth noticing here that in $\mathsf{letval}\ x = x\ \mathsf{in}\ k\ x$, the second "$x$" is also a free variable (and thus should also be included in an environment).

Figure 2 and Figure 3 demonstrate the mutual recursive function $\mathcal{F}$ and function $\mathcal{H}$ to calculate the free variables of a CPS term and CPS value respectively. Both $\mathcal{F}$ and $\mathcal{H}$ return an ordered set of variable names. Function `set`

generate a set of strings from a string list.

$$
\begin{aligned}
\mathcal{F} \ &: \ \text{Cps.t} \rightarrow \text{string set} \\
\mathcal{F}(\mathsf{letval}\ x = V\ \mathsf{in}\ K) &= (\mathcal{F}(K)/x) \cup \mathcal{H}(V) \\
\mathcal{F}(\mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K) &= (\mathcal{F}(K)/x) \cup \{y\} \\
\mathcal{F}(\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ K') &= (\mathcal{F}(K)/x) \cup (\mathcal{F}(K')/k) \\
\mathcal{F}(k\ x) &= \{k, x\} \\
\mathcal{F}(f\ k\ x) &= \{f, k, x\} \\
\mathcal{F}(\mathsf{case}\ x\ \mathsf{of}\ k_1 \parallel k_2) &= \{x, k_1, k_2\} \\
\mathcal{F}(\mathsf{letprim}\ x = PrimOp\ \vec{y}\ \ \mathtt{in}\ K) &= (\mathcal{F}(K)/x) \cup \mathtt{set}(\vec{y}) \\
\mathcal{F}(\mathsf{if0}\ x\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2) &= \{x, k_1, k_2\} \\
\mathcal{F}(\mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K') &= (\mathcal{F}(K) - \{f, k, x\}) \cup (\mathcal{F}(K')/f)
\end{aligned}
$$

Figure 2: Calculating Free Variables in CPS terms

$$
\begin{aligned}
\mathcal{H} \ &: \ \text{Cps.v} \rightarrow \text{string set} \\
\mathcal{H}(()) &= \emptyset \\
\mathcal{H}(i) &= \emptyset \\
\mathcal{H}(\text{"}s\text{"}) &= \emptyset \\
\mathcal{H}((x_1, x_2,\ \ldots\ , x_n)) &= \{x_1, x_2,\ \ldots\ , x_n\} \\
\mathcal{H}(\mathsf{in}_i\ x) &= \{x\} \\
\mathcal{H}(\lambda k\ x.K) &= \mathcal{F}(K) - \{k, x\}
\end{aligned}
$$

Figure 3: Calculating Free Variables in CPS values

Exercise 1. Read the formulas above carefully. Make sure you understand each one before moving on to the next section.

For example, consider this question: In the first rule:

$$\mathcal{F}(\mathsf{letval}\ x = V\ \mathsf{in}\ K) = (\mathcal{F}(K)/x) \cup \mathcal{H}(V),$$

can we remove x after the union of the two sets? Namely, will

$$\mathcal{F}(\mathsf{letval}\ x = V\ \mathsf{in}\ K) = (\mathcal{F}(K) \cup \mathcal{H}(V))/x$$

be correct?

3 Closure Syntax

In this lab, the target language is a closure-passing language. Wherever a function is needed to be returned as a result, or passed as an argument, a corresponding closure will be transmitted instead. Aside from that, the Closure syntax (shown in Figure 4) is similar to the CPS syntax.

(terms)	K	$\rightarrow$	$\mathsf{letval}\ x\ =\ V\ \ \mathsf{in}\ \ K$
		$\mid$	$\mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K$
		$\mid$	$\mathsf{letcont}\ k\ env\ x = K\ \mathsf{in}\ K'$
		$\mid$	$k\ env\ x$
		$\mid$	$f\ env\ k\ x$
		$\mid$	$\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow K \mid \mathsf{in}_2\ x_2 \Rightarrow K'$
		$\mid$	$\mathsf{letprim}\ x = PrimOp\ \vec{y}\ \ \mathtt{in}\ K$
		$\mid$	$\mathsf{if0}\ x\ \mathsf{then}\ K\ \mathsf{else}\ K'$
		$\mid$	$\mathsf{letfix}\ f\ env\ k\ x = K\ \mathsf{in}\ K'$
(values)	V	$\rightarrow$	$()$
		$\mid$	i
		$\mid$	$"_s"$
		$\mid$	$(x_1, x_2,\ \ldots\ , x_n)$
		$\mid$	$\mathsf{in}_i\ x$
		$\mid$	$\lambda env\ k\ x.K$
(primitives)	$PrimOp$	$\rightarrow$	$+$
		$\mid$	$-$
		$\mid$	$*$
		$\mid$	print
		$\mid$	$\mathsf{int2string}$

Figure 4: Closure syntax

Compared with the CPS syntax, there are basically two aspects of modifications:

- For function definitions, an extra formal parameter *env* is attached as the environment of a function:
  - $\mathsf{letcont}\ k\ \underline{env}\ x = K\ \mathsf{in}\ K'$
  - $\mathsf{letfix}\ f\ \underline{env}\ k\ x = K\ \mathsf{in}\ K'$
  - $\lambda\underline{env}\ k\ x.K$
- For function applications (continuations are treated as functions after the CPS phase), an extra actual parameter *env* is provided:
  - $k\ \underline{env}\ x$
  - $f\ \underline{env}\ k\ x$

Exercise 2. Finish the function `freeVarTerm` and `freeVarValue` in file `closure-convert.sml`. Function `freeVarTerm` calculate the free variables in a CPS term, and function `freeVarValue` calculate the free variables in a CPS value.

To rule out redundant calculation and for convenience sake, we store the free variables in closure syntax tree directly. Therefore during the calculation, the CPS syntax term is converted into a closure passing syntax tree carrying information about free variables. This translation is just a naive substitution without doing the actual closure conversion, thus will not carry correctness. We will do the actual closure conversion in the next section.

To encode this, function `freeVarTerm` and `freeVarValue` return a pair of the converted syntax item and the free variable set of the item, both of them are constructed recursively.

There are 3 kind of syntax items where a function will be defined:

$$\mathsf{letcont}\ k\ env\ x = K\ \mathsf{in}\ K'$$
$$\mathsf{letfix}\ f\ env\ k\ x = K\ \mathsf{in}\ K'$$
$$\lambda env\ k\ x.K.$$

Information about free variables of a function is stored where it is defined. You will find an extra component - a string list is attached in closure syntax definitions for the three items in our code.

Pay special attention that the free variable sets these three syntax items carrying only contain the free variables of *the function definition* (i.e., free variables of the function body with the arguments excluded), rather than of the entire syntax item. You'll understand this better in the next section.

The utility function `freshCont`, `freshEnv` and `freshVal` are provided to generate new exclusive names.

You can test your implementation with some examples and examine the printed free variable sets to check whether you have implemented the functions properly.

Exercise 3. Finish the code in file `closure.sml` to pretty print a `Closure.t`. However as we haven't dealt with types, the output is not likely to pass the compilation of SML/NJ.

You should print out the information of free variables stored in the closure syntax tree to check your `freeVarTerm` and `freeVarValue` function with some examples before continue.

4 Closure Conversion

In this section we will discuss how to perform closure conversion. The function Θ described in Figure 5 and Figure 6 translate a CPS term into a closure passing term.

While some CPS terms are translated easily with a recursive application of Θ, some other CPS terms call for special concern. The latter can be divided into a few categories:

1. Function definitions: $\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ K'$, $\mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K'$, and $\lambda k\ x.K$.

   Recalling what we have discussed in Section 1, there are some alterations for a syntax item where a function is defined:

   (a) A formal parameter representing the environment is inserted (the *env* field of the target syntax item.). [2]

   (b) In a function body, all free variables must be fetched out from the environment first. This is done by adding a series of nested `let` expression. After the previous section, the free variables we need are already stored in the syntax tree.

   (c) Before entering the "`in`" body (say K' of $\mathsf{letcont}\ k\ env\ x = K\ \mathsf{in}\ K'$), an actual environment should be generated, by inserting a `letval` expression that binds a variable with a tuple of the function's free variables. Because we are out of the function definition here, these variables are not "free" now, which means we can access them directly.

   Then the closure containing the function name and the environment variable is generated via another `letval` clause. Note that *the closure should have the same name with the original function*, which allows function applications in the "in" body to find the right closure for the right function.

   (d) The term $\mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K'$ requires special treatment in that the function f might be used inside its function body K. The problem is solved easily by inserting "`letval` $f = (f_{code}, env)$ `in` ..." outside K, to define a closure with the name f consisted of the (new) function name f_{code} and the parameter env.

2. Function applications: $k\ x$, $f\ k\ x$.

   To perform a function application, we need to fetch the function code and the environment from the corresponding closure (of the same name with the orignal function in CPS term), and apply the code to the environment and the original arguments.

   To unfold the closure, two `let` expressions are inserted.

[2] However you may want to finish this job during the free variable calculating phase, as the CPS terms are already converted to a closure syntax term there.

$$
\begin{aligned}
\Theta \;:\; & \text{Cps.t} \to \text{Closure.t} \\
\Theta(\mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K) &= \mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ K') &= \mathsf{letcont}\ k_{code}\ env\ x = \\
&\qquad \texttt{let}\ y_1 = \pi_1\ env\ \texttt{in} \\
&\qquad\quad \texttt{let}\ y_2 = \pi_2\ env\ \texttt{in} \\
&\qquad\quad \ldots \\
&\qquad\qquad \texttt{let}\ y_m = \pi_m\ env\ \texttt{in}\ \Theta(K) \\
&\qquad \mathsf{in\ letval}\ env' = (y_1, y_2, \ \ldots\ , y_m)\ \texttt{in} \\
&\qquad\quad \mathsf{letval}\ k = (k_{code}, env')\ \texttt{in}\ \Theta(K') \\
&\qquad (\mathit{where}\ \{y_1, \ \ldots\ , y_m\} = \mathcal{F}(K) - \{x\}) \\
\Theta(k\ x) &= \texttt{let}\ k_{code} = \pi_1\ k\ \texttt{in} \\
&\qquad \texttt{let}\ env = \pi_2\ k\ \texttt{in} \\
&\qquad\quad k_{code}\ env\ x \\
\Theta(f\ k\ x) &= \texttt{let}\ f_{code} = \pi_1\ f\ \texttt{in} \\
&\qquad \texttt{let}\ env = \pi_2\ f\ \texttt{in} \\
&\qquad\quad f_{code}\ env\ k\ x \\
\Theta(\mathsf{case}\ x\ \mathsf{of}\ k_1 \parallel k_2) &= \mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow \Theta(k_1\ x_1) \\
&\qquad\quad \mid \mathsf{in}_2\ x_2 \Rightarrow \Theta(k_2\ x_2) \\
\Theta(\mathsf{letprim}\ x = PrimOp\ \vec{y}\ \texttt{in}\ K) &= \mathsf{letprim}\ x = PrimOp\ \vec{y}\ \texttt{in}\ \Theta(K) \\
\Theta(\mathsf{if0}\ x\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2) &= \mathsf{letval}\ x_1 = ()\ \mathsf{in} \\
&\qquad \mathsf{if0}\ x\ \mathsf{then}\ \Theta(k_1\ x_1)\ \mathsf{else}\ \Theta(k_2\ x_1) \\
\Theta(\mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K') &= \mathsf{letfix}\ f_{code}\ env\ k\ x = \\
&\qquad \texttt{letval}\ f = (f_{code}, env)\ \texttt{in} \\
&\qquad\quad \texttt{let}\ y_1 = \pi_1\ env\ \texttt{in} \\
&\qquad\qquad \texttt{let}\ y_2 = \pi_2\ env\ \texttt{in} \\
&\qquad\qquad \ldots \\
&\qquad\qquad\quad \texttt{let}\ y_m = \pi_m\ env\ \texttt{in}\ \Theta(K) \\
&\qquad\quad \mathsf{in\ letval}\ env' = (y_1, y_2, \ \ldots\ , y_m)\ \texttt{in} \\
&\qquad\qquad \mathsf{letval}\ f = (f_{code}, env')\ \texttt{in}\ \Theta(K') \\
&\qquad (\mathit{where}\ \{y_1, \ \ldots\ , y_m\} = \mathcal{F}(K) - \{f, k, x\})
\end{aligned}
$$

Figure 5: Closure Conversion for CPS terms

Exercise 4. Finish the code in file `closure-convert.sml` for the function `convert` that performs the actual closure conversion. Before you start, make sure that you understand the rules in Figure 5 and 6 thoroughly.

You can test your implementation on the give example, or write some of your own examples.

$$
\begin{aligned}
\Theta \;:\; & \text{Cps.t} \to \text{Closure.t} \\
\Theta(\mathsf{letval}\ x = ()\ \mathsf{in}\ K) &= \mathsf{letval}\ x = ()\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letval}\ x = i\ \mathsf{in}\ K) &= \mathsf{letval}\ x = i\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letval}\ x = \text{"}s\text{"}\ \mathsf{in}\ K) &= \mathsf{letval}\ x = \text{"}s\text{"}\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letval}\ x = (x_1, x_2,\ \ldots\ , x_n)\ \mathsf{in}\ K) &= \mathsf{letval}\ x = (x_1, x_2,\ \ldots\ , x_n)\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letval}\ x = \mathsf{in}_i\ y\ \mathsf{in}\ K) &= \mathsf{letval}\ x = \mathsf{in}_i\ y\ \mathsf{in}\ \Theta(K) \\
\Theta(\mathsf{letval}\ x = \lambda k\ z.K\ \mathsf{in}\ K') &= \mathsf{letval}\ x_{code} = \lambda env\ k\ z. \\
&\qquad \texttt{let}\ y_1 = \pi_1\ env\ \texttt{in} \\
&\qquad\quad \texttt{let}\ y_2 = \pi_2\ env\ \texttt{in} \\
&\qquad\quad \ldots \\
&\qquad\qquad \texttt{let}\ y_m = \pi_m\ env\ \texttt{in}\ \Theta(K) \\
&\qquad \mathsf{in}\ \mathsf{letval}\ env' = (y_1, y_2,\ \ldots\ , y_m)\ \texttt{in} \\
&\qquad\quad \mathsf{letval}\ x = (x_{code}, env')\ \texttt{in}\ \Theta(K') \\
&\qquad (where\ \{y_1,\ \ldots\ , y_m\} = \mathcal{F}(K) - \{k, z\})
\end{aligned}
$$

Figure 6: Closure Conversion for CPS terms (continued)