Defunctionalization

Di Zhao

zhaodi01@mail.ustc.edu.cn

Document for the defunctionalization phase.

This stage will eliminate the lambda-absraction. The input is the CPS representaion, the output of this stage is the FLAT representation.

1 Calculating free variables

Still, the first step is to the free variable calculation. The input in this phase is the CPS representation (Figure 1).

$$
\begin{array}{rrcl}
\text{(terms)} & K & \to & \mathsf{letval}\ x\ =\ V\ \ \mathsf{in}\ \ K \\
 & & | & \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ K' \\
 & & | & k\ x \\
 & & | & f\ k\ x \\
 & & | & \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathtt{in}i_j\ x_j\ \mathtt{=>}\ K_j} \\
 & & | & \mathsf{letprim}\ x = PrimOp\ \vec{y}\ \mathtt{in}\ K \\
 & & | & \mathsf{if}\ x\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \\
 & & | & \mathsf{letfix}\ f\ k\ x = K\ \mathsf{in}\ K' \\
\text{(values)} & V & \to & ()\ \ |\ \mathsf{true}\ \ |\ \mathsf{false} \\
 & & | & i\ \ \ |\ \ \ "s" \\
 & & | & (x_1, x_2, ..., x_n) \\
 & & | & \mathsf{in}i\ x \\
 & & | & \lambda k\ x.K \\
 & & | & \mathsf{\#}i\ x \\
\text{(primitive} & PrimOp & \to & +\ |\ -\ |\ * \\
\text{operations)} & & | & >\ |\ <\ |\ = \\
 & & | & \mathsf{andalso}\ \ |\ \mathsf{orelse}\ \ |\ \ \mathsf{not} \\
 & & | & \mathsf{print}\ |\ \mathsf{int2string}
\end{array}
$$

Figure 1: CPS syntax

The procedure is identical will that in the closure conversion . The functions are illustrated in Figure 2 and Figure 3.

(The code should be reusable. However as we put the free variable information in the closure syntax tree earlier, so the function code cannot be reused directly. Maybe its better to store that in the CPS syntax tree.)

$$
\begin{aligned}
\mathcal{F} &: \text{Cps.t} \rightarrow \text{string set} \\
\mathcal{F}(\textsf{letval } x = V \textsf{ in } K) &= (\mathcal{F}(K)/x) \cup \mathcal{H}(V) \\
\mathcal{F}(\textsf{letcont } k\ x = K \textsf{ in } K') &= (\mathcal{F}(K)/x) \cup (\mathcal{F}(K')/k) \\
\mathcal{F}(k\ x) &= \{k, x\} \\
\mathcal{F}(f\ k\ x) &= \{f, k, x\} \\
\mathcal{F}(\textsf{case } x \textsf{ of } \overrightarrow{\texttt{in}i_j\ x_j \texttt{ => } K_j}) &= \{x\} \cup (\mathcal{F}(K_1)/x_1) \cup \ldots \cup (\mathcal{F}(K_n)/x_n) \\
&\quad (\text{where } j = 1,\ \ldots\ , n) \\
\mathcal{F}(\textsf{letprim } x = PrimOp\ \vec{y} \textsf{ in } K) &= (\mathcal{F}(K)/x) \cup \texttt{set}(\vec{y}) \\
\mathcal{F}(\textsf{if } x \textsf{ then } k_1 \textsf{ else } k_2) &= \{x, k_1, k_2\} \\
\mathcal{F}(\textsf{letfix } f\ k\ x = K \textsf{ in } K') &= (\mathcal{F}(K) - \{f, k, x\}) \cup (\mathcal{F}(K')/f)
\end{aligned}
$$

Figure 2: Calculating Free Variables in CPS terms

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

Figure 3: Calculating Free Variables in CPS values

2 Target Syntax

The final output of this stage is a defunctionalized syntax. We name it as Defunc. The syntax is shown in Figure 4.

A program consists of two functions: *applycont*, $applyfunc$ and an expression. The function $applyfunc$ takes three arguments (f, k, x) and its body is a case expression whose condition is f. Similarly, The function *applycont* takes two arguments (k, x) and its body is a case expression whose condition is k.

As for terms (expressions), there won't be any abstractions or function definitions. Function definitions are replaced by tagged values and the code will be inserted into the *applycont* or $applyfunc$ function.

$$
\begin{array}{rrcl}
\text{(program)} & p & \rightarrow & \mathit{applyfunc}(f,k,x)\{K_1\} \\
 & & & \mathit{applycont}(k_1,x_1)\{K_2\} \\
 & & & K_3 \\
\text{(terms)} & K,T & \rightarrow & \mathsf{letval}\ x\ =\ V\ \ \mathsf{in}\ \ K \\
 & & | & \mathit{applycont}(k,\ x) \\
 & & | & \mathit{applyfunc}(f,\ k,\ x) \\
 & & | & \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in}i_j\ x_j \Rightarrow K_j} \\
 & & | & \mathsf{letprim}\ x = \mathit{PrimOp}\ \vec{y}\ \mathsf{in}\ K \\
 & & | & \mathsf{if}\ x\ \mathsf{then}\ K\ \mathsf{else}\ K' \\
\text{(values)} & V & \rightarrow & ()\ \ |\ \mathsf{true}\ |\ \mathsf{false} \\
 & & | & i\quad |\quad ''s'' \\
 & & | & (x_1,x_2,...,x_n) \\
 & & | & \mathsf{in}i\ x \\
 & & | & \lambda k\ x.K \\
 & & | & \#i\ x \\
\text{(primitive} & \mathit{PrimOp} & \rightarrow & +\ |\ -\ |\ * \\
\text{operations)} & & | & >\ |\ <\ |\ = \\
 & & | & \mathsf{andalso}\ \ |\ \mathsf{orelse}\ \ |\ \mathsf{not} \\
 & & | & \mathsf{print}\ |\ \mathsf{int2string}
\end{array}
$$

Figure 4: Defunc syntax

3 Defunctionalization

The functions performing defunctionalization is defined in Figure 5, 6, 7. The functions translate CPS syntax into Defunc syntax by adding case branches into the function *applyfunc* and *applycont*, to represent function definitions.

Function $\mathsf{replace}(x, y, K)$ replaces the free existences of x with y in term K.

$$\frac{\begin{array}{l} K_1, \mathit{applyfunc}(fa,ka,xa)\{K_3\}, \mathit{applycont}(ka',xa')\{K_4\} \rightsquigarrow \\ \qquad K_1', \mathit{applyfunc}(fa,ka,xa)\{K_5\}, \mathit{applycont}(ka',xa')\{K_6\} \\ K_2, \mathit{applyfunc}(fa,ka,xa)\{K_5\}, \mathit{applycont}(ka',xa')\{K_6\} \rightsquigarrow \\ \qquad K_2', \mathit{applyfunc}(fa,ka,xa)\{K_7\}, \mathit{applycont}(ka',xa')\{\mathsf{case}\ ka'\ \mathsf{of}\ \vec{b}\} \end{array}}{\begin{array}{l} \mathsf{letcont}\ k\ x = K_1\ \mathsf{in}\ K_2, \mathit{applyfunc}(fa,ka,xa)\{K_3\}, \mathit{applycont}(ka',xa')\{K_4\} \rightsquigarrow \\ \qquad \mathsf{letval}\ env = (y_1,\ y_2,\ \dots,\ y_m)\ \mathsf{in}\ \mathsf{letval}\ k = \mathtt{in}i\ env\ \mathsf{in}\ K_2', \\ \qquad \mathit{applyfunc}(fa,ka,xa)\{K_7\}, \\ \qquad \mathit{applycont}(ka',xa')\{\mathsf{case}\ ka'\ \mathsf{of}\ \mathsf{in}i\ env \Rightarrow \mathsf{letval}\ y_1 = \#1\ env\ \mathsf{in} \\ \qquad\qquad \mathsf{letval}\ y_2 = \#2\ env\ \mathsf{in} \\ \qquad\qquad \dots \\ \qquad\qquad \mathsf{letval}\ y_m = \#m\ env\ \mathsf{in} \\ \qquad\qquad \mathsf{replace}(x, xa', K_1') \\ \qquad :: \vec{b}\} \\ \qquad \text{(where } i \text{ is a freshed tag value and } \{y_1,\ y_2,\ \dots,\ y_m\} = \mathcal{F}(K_1)/x) \end{array}}$$

$$\frac{}{\begin{array}{l} k\ x, \mathit{applyfunc}(fa,ka,xa)\{K_1\}, \mathit{applycont}(ka',xa')\{K_2\} \rightsquigarrow \\ \qquad \mathit{applycont}(k,x), \mathit{applyfunc}(fa,ka,xa)\{K_1\}, \mathit{applycont}(ka',xa')\{K_2\} \end{array}}$$

$$\frac{}{\begin{array}{l} f\ k\ x, \mathit{applyfunc}(fa,ka,xa)\{K_1\}, \mathit{applycont}(ka',xa')\{K_2\} \rightsquigarrow \\ \qquad \mathit{applyfunc}(f,k,x), \mathit{applyfunc}(fa,ka,xa)\{K_1\}, \mathit{applycont}(ka',xa')\{K_2\} \end{array}}$$

$$\frac{\begin{array}{l} K_1, \mathit{applyfunc}(fa,ka,xa)\{T\}, \mathit{applycont}(ka',xa')\{T'\} \rightsquigarrow \\ \qquad K_1', \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_1'\} \\ K_2, \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_1'\} \rightsquigarrow \\ \qquad K_2', \mathit{applyfunc}(fa,ka,xa)\{T_2\}, \mathit{applycont}(ka',xa')\{T_2'\} \\ \qquad \dots \\ K_m, \mathit{applyfunc}(fa,ka,xa)\{T_{m-1}\}, \mathit{applycont}(ka',xa')\{T_{m-1}'\} \rightsquigarrow \\ \qquad K_m', \mathit{applyfunc}(fa,ka,xa)\{T_m\}, \mathit{applycont}(ka',xa')\{T_m'\} \end{array}}{\begin{array}{l} \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathtt{in}i_j\ x_j\ \texttt{=>}\ K_j}, \mathit{applyfunc}(fa,ka,xa)\{T\}, \mathit{applycont}(ka',xa')\{T'\} \rightsquigarrow \\ \qquad \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathtt{in}i_j\ x_j\ \texttt{=>}\ K_j'}, \mathit{applyfunc}(fa,ka,xa)\{T_m\}, \mathit{applycont}(ka',xa')\{T_m'\} \\ \qquad \text{(where } j = 1,\ 2,\ \dots,\ m\text{)} \end{array}}$$

$$\frac{\begin{array}{l} K, \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_2\} \rightsquigarrow \\ \qquad K_1, \mathit{applyfunc}(fa,ka,xa)\{T_3\}, \mathit{applycont}(ka',xa')\{T_4\} \end{array}}{\begin{array}{l} \mathsf{letprim}\ x = \mathit{PrimOp}\ \vec{y}\ \mathtt{in}\ K, \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_2\} \rightsquigarrow \\ \qquad \mathsf{letprim}\ x = \mathit{PrimOp}\ \vec{y}\ \mathtt{in}\ K_1, \mathit{applyfunc}(fa,ka,xa)\{T_3\}, \mathit{applycont}(ka',xa')\{T_4\} \end{array}}$$

$$\frac{}{\begin{array}{l} \mathsf{if}\ x\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2, \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_2\} \rightsquigarrow \\ \qquad \mathsf{letval}\ x_1 = ()\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \mathit{applycont}(k_1\ x_1)\ \mathsf{else}\ \mathit{applycont}(k_2\ x_1), \\ \qquad \mathit{applyfunc}(fa,ka,xa)\{T_1\}, \mathit{applycont}(ka',xa')\{T_2\} \end{array}}$$

Figure 5: Defunctionalization for CPS terms

$$\frac{\begin{array}{l} K_1, applyfunc(fa,ka,xa)\{K_3\}, applycont(ka_1,xa_1)\{K_4\} \leadsto \\ \qquad K_1', applyfunc(fa,ka,xa)\{K_5\}, applycont(ka',xa')\{K_6\} \\ K_2, applyfunc(fa,ka,xa)\{K_5\}, applycont(ka',xa')\{K_6\} \leadsto \\ \qquad K_2', applyfunc(fa,ka,xa)\{\mathsf{case}\ fa\ \mathsf{of}\ \vec{b}\}, applycont(ka',xa')\{K_7\} \end{array}}{\mathsf{letfix}\ f\ k\ x = K_1\ \mathsf{in}\ K_2, applyfunc(fa,ka,xa)\{K_3\}, applycont(ka_1,xa_1)\{K_4\} \leadsto}$$

$$\begin{array}{l} \mathsf{letval}\ env = (y_1,\ y_2,\ \ldots,\ y_m)\ \mathsf{in}\ \mathsf{letval}\ f = \mathtt{in}i\ env\ \mathsf{in}\ K_2', \\ applyfunc(fa,ka,xa)\{\mathsf{case}\ fa\ \mathsf{of}\ \mathsf{in}i\ env \Rightarrow \begin{array}[t]{l} \mathsf{letval}\ f = \mathtt{in}i\ env\ \mathsf{in} \\ \mathsf{letval}\ y_1 = \#1\ env\ \mathsf{in} \\ \mathsf{letval}\ y_2 = \#2\ env\ \mathsf{in} \\ \quad \ldots \\ \mathsf{letval}\ y_m = \#m\ env\ \mathsf{in} \\ \mathsf{replace}(k,ka,\mathsf{replace}(x,xa,K_1')) \end{array} \\ \qquad\qquad\qquad\qquad :: \vec{b}\}, \\ applycont(ka',xa')\{K_7\} \end{array}$$

(where i is a freshed tag value and $\{y_1,\ y_2,\ \ldots,\ y_m\} = \mathcal{F}(K_1) - \{f,k,x\}$)

$$\frac{\begin{array}{l} K_1, applyfunc(fa,ka,xa)\{K_3\}, applycont(ka_1,xa_1)\{K_4\} \leadsto \\ \qquad K_1', applyfunc(fa,ka,xa)\{K_5\}, applycont(ka',xa')\{K_6\} \\ K_2, applyfunc(fa,ka,xa)\{K_5\}, applycont(ka',xa')\{K_6\} \leadsto \\ \qquad K_2', applyfunc(fa,ka,xa)\{\mathsf{case}\ fa\ \mathsf{of}\ \vec{b}\}, applycont(ka',xa')\{K_7\} \end{array}}{\mathsf{letval}\ x = \lambda k\ z.K_1\ \mathsf{in}\ K_2, applyfunc(fa,ka,xa)\{K_3\}, applycont(ka_1,xa_1)\{K_4\} \leadsto}$$

$$\begin{array}{l} \mathsf{letval}\ env = (y_1,\ y_2,\ \ldots,\ y_m)\ \mathsf{in}\ \mathsf{letval}\ x = \mathtt{in}i\ env\ \mathsf{in}\ K_2', \\ applyfunc(fa,ka,xa)\{\mathsf{case}\ fa\ \mathsf{of}\ \mathsf{in}i\ env \Rightarrow \begin{array}[t]{l} \mathsf{letval}\ y_1 = \#1\ env\ \mathsf{in} \\ \mathsf{letval}\ y_2 = \#2\ env\ \mathsf{in} \\ \quad \ldots \\ \mathsf{letval}\ y_m = \#m\ env\ \mathsf{in} \\ \mathsf{replace}(k,ka,\mathsf{replace}(z,xa,K_1')) \end{array} \\ \qquad\qquad\qquad\qquad :: \vec{b}\}, \\ applycont(ka',xa')\{K_7\} \end{array}$$

(where i is a freshed tag value and $\{y_1,\ y_2,\ \ldots,\ y_m\} = \mathcal{F}(K_1) - \{k,z\}$)

$$\frac{\begin{array}{l} K, applyfunc(fa,ka,xa)\{T_1\}, applycont(ka',xa')\{T_2\} \leadsto \\ \qquad K_1, applyfunc(fa,ka,xa)\{T_3\}, applycont(ka',xa')\{T_4\} \end{array}}{\begin{array}{l} \mathsf{letval}\ x = V\ \ \mathtt{in}\ K, applyfunc(fa,ka,xa)\{T_1\}, applycont(ka',xa')\{T_2\} \leadsto \\ \qquad \mathsf{letval}\ x = V\ \ \mathtt{in}\ K_1, applyfunc(fa,ka,xa)\{T_3\}, applycont(ka',xa')\{T_4\} \end{array}}$$

(given that V is not a lambda abstraction.)

Figure 6: Defunctionalization for CPS terms(continued)

4 Code generation

The code generation process for the defunctionalized syntax tree is intuitive. One thing to be noticed is that the `case` expression may contain a lot of bindings (in contrast, if we perform code generation following the closure conversion process, only a tuple may be defined there in case of function calls). In this way, the C code for each switch case should be in a block (surrounded by a pair of "{}").

Moreover, as the *appl_func* function and *apply_cont* function may be mutually recursive, we may need to output the function declarations for them first.

The modifications should also be included when generation code for garbage collection.

Updates:

15-9-5: Added boolean values and corresponding operations. Changed `if0` expression into `if` expression.