CPS Conversion

Di Zhao

zhaodi01@mail.ustc.edu.cn

Document related to the CPS conversion phase.

1 ML Syntax

$$
\begin{array}{rcll}
e & \to & x & \textit{variable} \\
 & | & \texttt{()} & \textit{empty} \\
 & | & \texttt{true} & \textit{true} \\
 & | & \texttt{false} & \textit{false} \\
 & | & i & \textit{integer} \\
 & | & "s" & \textit{string} \\
 & | & \texttt{fn}\ x\ \texttt{=>}\ e & \textit{abstraction} \\
 & | & e_1\ e_2 & \textit{application} \\
 & | & (e_1, e_2, ..., e_n) & \textit{tuple} \\
 & | & \texttt{\#}i\ e & \textit{projection} \\
 & | & \mathsf{in}i\ e & \textit{tagged value} \\
 & | & \mathsf{case}\ e\ \mathsf{of}\ \overrightarrow{\texttt{in}i_j\ x_j\ \texttt{=>}\ e_j} & \textit{case} \\
 & | & PrimOp\ \vec{e} & \textit{operations} \\
 & | & \mathsf{let\ val}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} & \textit{let} \\
 & | & \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \textit{if} \\
 & | & \mathsf{let\ fix}\ f\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} & \textit{fix} \\
PrimOp & \to & + & \textit{add} \\
 & | & - & \textit{sub} \\
 & | & * & \textit{times} \\
 & | & > & \textit{less than} \\
 & | & < & \textit{larger than} \\
 & | & = & \textit{equals} \\
 & | & \texttt{not} & \textit{not} \\
 & | & \texttt{andalso} & \textit{and also} \\
 & | & \texttt{orelse} & \textit{or else} \\
 & | & \texttt{print} & \textit{print} \\
 & | & \texttt{int2string} & \textit{toString}
\end{array}
$$

Figure 1: ML Syntax

Figure 1 illustrates the syntax of our source language - a subset of ML. Here we use the metavariable e to represent an arbitrary expression of the source language. Similarly, x is a metavariable ranging over variables.

Updates:

15-7-1: The case expression is changed to: case e of $\overrightarrow{\texttt{in}i_j\ x_j\ \texttt{=>}\ e_j}$, to expand the original dualistic cases into indefinite cases, to facilitate generating the `apply` function in the defunctionalization phase. i is the integer representing the type constructor. We need a front end to map the constructors in `datatype` definitions with integers.

We may **need a typing system** to generated executive ML code for each intermediate representation. (We don't know how many labels there are.)

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

2 CPS Syntax

Figure 2 illustrates the syntax of the CPS language corresponding to the ML syntax in Figure 1. In the CPS syntax, we introduce the metavariable k to represent a continuation.

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

Figure 2: CPS syntax

Updates:

15-7-1: Removed `let` $x = \pi_i\ y$ `in` K to simplify the syntax and rules. Instead, add `#`$i\ x$ to the values.

Change the form of `case` to enable multiple cases (and to make the transformations look better?).

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

3 CPS Conversion

In this section we will discuss how to perform CPS conversion. Expressions in ML can be translated into untyped CPS terms using the function shown in Figure 3. This is an adaptation of the standard higher-order one-pass call-by-value transformation (Danvy and Filinski 1992).

Updates:

15-7-2: Modified the conversion rule for `case` to enable multiple cases. Modified the rule for projection operation (see updates for 15-7-1).

15-9-5: Added conversion rules for boolean values. Changed `if0` expression into `if` expression.

$$
\begin{aligned}
[\![\cdot]\!] &: \mathrm{ML} \to (\mathrm{Var} \to \mathrm{CTm}) \to \mathrm{CTm} \\
[\![x]\!]\kappa &= \kappa(x) \\
[\![\texttt{()}]\!]\kappa &= \texttt{letval}\ x = ()\ \texttt{in}\ \kappa(x) \\
[\![\texttt{true}]\!]\kappa &= \texttt{letval}\ x = \texttt{true}\ \texttt{in}\ \kappa(x) \\
[\![\texttt{false}]\!]\kappa &= \texttt{letval}\ x = \texttt{false}\ \texttt{in}\ \kappa(x) \\
[\![i]\!]\kappa &= \texttt{letval}\ x = i\ \texttt{in}\ \kappa(x) \\
[\![\text{"}s\text{"}]\!]\kappa &= \texttt{letval}\ x = \text{"}s\text{"}\ \texttt{in}\ \kappa(x) \\
[\![e_1\ e_2]\!]\kappa &= [\![e_1]\!](\Lambda z_1.[\![e_2]\!](\Lambda z_2.\texttt{letcont}\ k\ x = \kappa(x)\ \texttt{in}\ z_1\ k\ z_2)) \\
[\![(e_1, ..., e_n)]\!]\kappa &= (\![[e_1, ..., e_n], \texttt{nil}]\!)(\Lambda \vec{l}. \\
&\qquad \texttt{letval}\ x = \texttt{tuple}(\vec{l})\ \texttt{in}\ \kappa(x)) \\
[\![\texttt{in}i\ e]\!]\kappa &= [\![e]\!](\Lambda z.\texttt{letval}\ x = \texttt{in}i\ z\ \texttt{in}\ \kappa(x)) \\
[\![\texttt{\#}i\ e]\!]\kappa &= [\![e]\!](\Lambda z.\texttt{letval}\ x = \texttt{\#}i\ z\ \texttt{in}\ \kappa(x)) \\
[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!]\kappa &= \texttt{letval}\ f = \lambda k\ x.[\![e]\!](\Lambda z.k\ z)\ \texttt{in}\ \kappa(f) \\
[\![\texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\kappa &= \texttt{letcont}\ j\ x = [\![e_2]\!]\ \kappa\ \texttt{in}\ [\![e_1]\!](\Lambda z.j\ z) \\
[\![\texttt{case}\ e\ \texttt{of}\ \overrightarrow{\texttt{in}i_j\ x_j\ \texttt{=>}\ e_j}]\!]\kappa &= [\![e]\!](\Lambda z.\texttt{letcont}\ k_0\ x_0 = \kappa(x_0)\ \texttt{in} \\
&\qquad \texttt{letcont}\ k_1\ x_1 = [\![e_1]\!](\Lambda z.k_0\ z)\ \texttt{in} \\
&\qquad\ \ \texttt{letcont}\ k_2\ x_2 = [\![e_2]\!](\Lambda z.k_0\ z)\ \texttt{in} \\
&\qquad\qquad ... \\
&\qquad\qquad\ \ \texttt{letcont}\ k_n\ x_n = [\![e_n]\!](\Lambda z.k_0\ z)\ \texttt{in} \\
&\qquad\qquad\quad \texttt{case}\ z\ \texttt{of}\ \overrightarrow{\texttt{in}i_j\ y_j\ \texttt{=>}\ k_j\ y_j}) \\
&\text{(where } j = 1, 2,\ ...\ , n) \\
[\![\mathit{PrimOp}\ \vec{e}\ ]\!]\kappa &= (\![\vec{e}, \texttt{nil}]\!)(\Lambda \vec{l}.\texttt{letprim}\ x = \mathit{PrimOp}\ \vec{l}\ \texttt{in}\ \kappa(x))) \\
[\![\texttt{let fix}\ f\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]\kappa &= \texttt{letfix}\ f\ k\ x = [\![e_1]\!](\Lambda z.k\ z)\ \texttt{in}\ [\![e_2]\!]\kappa \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\kappa &= [\![e_1]\!](\Lambda z.\texttt{letcont}\ k_0\ x_0 = \kappa(x_0)\ \texttt{in} \\
&\qquad \texttt{letcont}\ k_1\ x_1 = [\![e_2]\!](\Lambda z.k_0\ z)\ \texttt{in} \\
&\qquad\ \ \texttt{letcont}\ k_2\ x_2 = [\![e_3]\!](\Lambda z.k_0\ z)\ \texttt{in} \\
&\qquad\qquad \texttt{if}\ z\ \texttt{then}\ k_1\ \texttt{else}\ k_2)
\end{aligned}
$$

Figure 3: CPS Conversion

$$
\begin{aligned}
(\!|\cdot|\!) &: \text{ ML list * string list} \to (\text{string list} \to \text{CTm}) \to \text{CTm} \\
(\!|\,\texttt{[]},\omega|\!)\eta &= \eta(\texttt{rev}(\omega)) \\
(\!|e :: es,\omega|\!)\eta &= [\![e]\!](\Lambda x.(\!|es, x :: \omega|\!)\eta)
\end{aligned}
$$

Figure 4: CPS Conversion for tuples