Hoisting, Allocation and Code Generation

Di Zhao

zhaodi01@mail.ustc.edu.cn

Document for the hoisting and allocation phases.

1 Hoisting

Hoist all the function definitions to top-level. Source language : closure passing language.

(terms)	K	$\rightarrow$	letval $x = V$ in K
		$\mid$	let $x =$ #i y in K
		$\mid$	letcont k env $x = K$ in K'
		$\mid$	k env x
		$\mid$	f env k x
		$\mid$	case x of $\overrightarrow{\text{in}i_j\ x_j \Rightarrow K_j}$
		$\mid$	letprim $x = PrimOp$ $\vec{y}$ in K
		$\mid$	if x then K else K'
		$\mid$	letfix f env k $x = K$ in K'
(values)	V	$\rightarrow$	()
		$\mid$	`true`
		$\mid$	`false`
		$\mid$	i
		$\mid$	"s"
		$\mid$	$(x_1, x_2, \ \ldots\ , x_n)$
		$\mid$	in$i$ x
		$\mid$	$\lambda env\ k\ x.K$
(primitive operations)	$PrimOp$	$\rightarrow$	$+ \mid - \mid *$
		$\mid$	$> \mid < \mid =$
		$\mid$	andalso $\mid$ orelse $\mid$ not
		$\mid$	print $\mid$ int2string

Figure 1: Closure syntax

The target language in this section is called Flat. The syntax is shown in Figure 2.

In the Flag language, a program p consists of a main function (denoted as m) and a list of functions ($\vec{f}$).

(program)	p	$\rightarrow$	$m;\ \vec{f}$
(functions)	$m,\ f$	$\rightarrow$	$x\ \vec{y}\ \{\ \vec{b};\ \ e;\ \}$
(bindings)	b	$\rightarrow$	$x = v$
(values)	v	$\rightarrow$	$()$
		$\mid$	`true`
		$\mid$	`false`
		$\mid$	i
		$\mid$	"s"
		$\mid$	$\pi_i\ x$
		$\mid$	$(x_1, x_2,\ \ldots\ , x_n)$
		$\mid$	$\mathsf{in}_i\ x$
		$\mid$	$PrimOp\ \vec{x}$
(primitive operations)	$PrimOp$	$\rightarrow$	$+\ \mid\ -\ \mid\ *$
		$\mid$	$>\ \mid\ <\ \mid\ =$
		$\mid$	andalso $\mid$ orelse $\mid$ not
		$\mid$	print $\mid$ int2string
(transfers)	e	$\rightarrow$	$x\ \vec{y}$
		$\mid$	`if` x `then` e_1 `else` e_2
		$\mid$	case x of $\overrightarrow{\mathsf{in}i_j\ x_j \Rightarrow e_j}$

Figure 2: Flat syntax

In a function definition $x\ \vec{y}\ \{\ \vec{b};\ \ e;\ \}$, x is the function name, $\vec{y}$ is the list of arguments. The function body consists of a list of bindings $\vec{b}$ and one control transferring expression e.

A binding binds a value v to a name x. Here the values to be named include empty value, constant integers, constant strings, projection operation, tuples, tagged values and primary operations.

There're three kinds of control transferring expressions (denoted as e).

The hoisting rules are illustrated in Figure 3.

Updates:

15-7-3: Change the form of `case` in `flat.sig`, `flat.sml`, and changed the respective hoisting rule in `hoist.sml` to enable multiple cases .

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

2 Allocation

After the allocation process, the allocation and initialization of tuples and tagged values will be explicit. Besides, the arguments of a single function are packed

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{letval}\ x = \lambda env\ k\ z.K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_x,\ \vec{b_2},\ e_2)} \text{ (H-LetValFunc)}$$
where f_x is function: $x\ [env, k, z]\{\vec{b_1};\ e_1;\}$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K \rightsquigarrow (\vec{f}, (x = V) :: \vec{b}, e)} \text{ (H-LetVal)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K \rightsquigarrow (\vec{f}, (x = \pi_i\ y) :: \vec{b}, e)} \text{ (H-Let)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{lecont}\ k\ env\ x = K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_k,\ \vec{b_2},\ e_2)} \text{ (H-LetCont)}$$
where f_k is function: $k\ [env, x]\{\vec{b_1};\ e_1;\}$

$$\frac{}{\vdash k\ env\ x \rightsquigarrow ([\,], [\,], k\ [env, x])} \text{ (H-ContApp)}$$

$$\frac{}{\vdash f\ env\ k\ x \rightsquigarrow ([\,], [\,], f\ [env, k, x])} \text{ (H-FuncApp)}$$

$$\frac{\vdash K_j \rightsquigarrow (\vec{f_j}, \vec{b_j}, e_j)}{\vdash \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in} i_j\ x_j \Rightarrow K_j} \rightsquigarrow (\vec{f_1} :: ... :: \vec{f_n},\ \vec{b_1} :: ... :: \vec{b_n},\ \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in} i_j\ x_j \Rightarrow e_j})} \text{ (H-Case)}$$
where $j = 1,\ ...\ , n$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{letprim}\ x = PrimOp\ \vec{y}\ \ \mathtt{in}\ K \rightsquigarrow (\vec{f}, (x, PrimOp\ \vec{y}) :: \vec{b}, e)} \text{ (H-LetPrim)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{if}\ x\ \mathsf{then}\ K\ \mathsf{else}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2},\ \vec{b_1} :: \vec{b_2},\ \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)} \text{ (H-If)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{letfix}\ f\ env\ k\ x = K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_f,\ \vec{b_2},\ e_2)} \text{ (H-LetFix)}$$
where f_f is function: $f\ [env, k, x]\{\vec{b_1};\ e_1;\}$

Figure 3: Hoisting rules for Closure.t

into one single argument, so that it would be easier to scan over the arguments when performing garbage collection. The target language in this phase is called: Machine. The syntax of Machine language is shown in Figure 4.

Given the Machine syntax, we can decide the conversion rules intuitively, as shown in Figure 5, 6, 7.

(program)	P	$\rightarrow$	$m;\ \vec{f}$
(functions)	$m,\ f$	$\rightarrow$	$x\ y\ \{\ \overrightarrow{binding};\ \ b;\ \}$
(bindings)	$binding$	$\rightarrow$	$x = v$
		$\mid$	$dst\texttt{[}i\texttt{]} = src$
(values)	v	$\rightarrow$	`null`
		$\mid$	`true`
		$\mid$	`false`
		$\mid$	i
		$\mid$	"s"
		$\mid$	$x\texttt{[}i\texttt{]}$
		$\mid$	`allocTuple`(i)
		$\mid$	`allocTag`(i)
		$\mid$	$PrimOp\ \vec{x}$
(primitive operations)	$PrimOp$	$\rightarrow$	$+\ \mid\ -\ \mid\ *$
		$\mid$	$>\ \mid\ <\ \mid\ =$
		$\mid$	andalso $\mid$ orelse $\mid$ not
		$\mid$	print $\mid$ int2string
(blocks)	b	$\rightarrow$	$\overrightarrow{binding};\ \ e$
(transfers)	e	$\rightarrow$	$x\ y$
		$\mid$	`if` x `then` b_1 `else` b_2
		$\mid$	case x of $\overrightarrow{\text{in}i_j\ x_j \Rightarrow b_j}$

Figure 4: Machine syntax

As for bindings, because tuples and tagged values are structured data elements, we need to allocate space for them and perform some initializations. In this way, both of them require special treatment:

- For $x = (x_1, ..., x_n)$, allocate a tuple x with n components and initialize component $x\texttt{[}i\texttt{]}$ with x_i.
- For $x = in_i\ y$, allocate structure x with tag value i and initialize component $x\texttt{[1]}$ with y.

Arguments of a function call now need to be packed into a tuple, and the tuple needs to be initialized by these arguments. In this way, control transferring expressions will be translated into a *block* which consists of a list of bindings and an expression.

For the same reason above, in a function definition, the original arguments need to be fetched out from the sole parameter of the function. Namely we need to add some bindings ahead of the original function body.

Pay attention that as the main function m takes no arguments, no additional bindings for argument folding is needed.

Updates:

$$
\begin{aligned}
\mathcal{A}_b : \text{string} * \text{Flat.binding} &\rightarrow \text{Machine.binding list} \\
\mathcal{A}_b(x = ()) &= [x = \texttt{null}] \\
\mathcal{A}_b(x = \texttt{true}) &= [x = \texttt{true}] \\
\mathcal{A}_b(x = \texttt{false}) &= [x = \texttt{false}] \\
\mathcal{A}_b(x = i) &= [x = i] \\
\mathcal{A}_b(x = \text{"}s\text{"}) &= [x = \text{"}s\text{"}] \\
\mathcal{A}_b(x = \pi_i\ y) &= [x = y\texttt{[}i\texttt{]}] \\
\mathcal{A}_b(x = (x_1, x_2,\ \ldots\ , x_n)) &= [x = \texttt{allocTuple}(n), \\
&\quad\ x\texttt{[1]} = x_1, \\
&\quad\ \ldots \\
&\quad\ x\texttt{[}n\texttt{]} = x_n] \\
\mathcal{A}_b(x = \mathsf{in}_i\ y) &= [x = \texttt{allocTag}(i), \\
&\quad\ x\texttt{[1]} = y] \\
\mathcal{A}_b(x = PrimOp\ \vec{y}) &= [x = PrimOp\ \vec{y}]
\end{aligned}
$$

Figure 5: Conversion from Flat to Machine for bindings

$$
\begin{aligned}
\mathcal{A}_e : \text{Flat.e} &\rightarrow \text{Machine.block} \\
\mathcal{A}_e(x\ \vec{y}) &= [z = \texttt{allocTuple}(n), \\
&\quad\ z\texttt{[1]} = y_1, \\
&\quad\ \ldots \\
&\quad\ z\texttt{[}n\texttt{]} = y_n]; \\
&\quad\ x\ z \\
&\quad\ (\text{given that } \vec{y} = [y_1,\ \ldots\ , y_n]) \\
\mathcal{A}_e(\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2) &= [\,]; \\
&\quad\ \texttt{if}\ x\ \texttt{then}\ \mathcal{A}_e(e_1)\ \texttt{else}\ \mathcal{A}_e(e_2) \\
\mathcal{A}_e(\mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in}i_j\ x_j \Rightarrow e_j}) &= [\,]; \\
&\quad\ \mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in}i_j\ x_j \Rightarrow \mathcal{A}_e(e_j)}
\end{aligned}
$$

Figure 6: Conversion from Flat to Machine for expressions

15-7-4: Change the form of `case` in `machine.sig`, `machine.sml`, and changed the respective allocation rule in `codegen.sml` to enable multiple cases .

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

$$
\begin{aligned}
\mathcal{A}_f(x\ \vec{y}\ \{\ \vec{b};\ e;\}) = x\ (z)\ \{ & \\
& \overrightarrow{y_i = z[i]} :: \\
& \overrightarrow{\mathcal{A}_b(b)}; \\
& \mathcal{A}_e(e); \\
\} & \\
\text{(given that } \vec{y} = [y_1,\ \ldots, y_n],\ i = 1,\ \ldots, n) &
\end{aligned}
$$

Figure 7: Conversion from Flat to Machine for functions

3 Code Generation

Code generation procedure: encode the functions to output the Machine syntax tree into a `.c` file which can be compiled by a C compiler and then be executed.

As is shown Figure 4, the Machine syntax is basically C style syntax. Consequently we can output C code for most of the syntax terms directly without making much modifications. However, there are a few aspects to which you may pay attention:

- Type and type-casting. As C is strong typed, you'll have to declare variables and insert type-castings with explicit types. For example, you can declare all the variables with `int*` type; when performing a function call, you may cast the function name into `void * (*)()` type first.
- Bindings of tuples and tagged values are encoded as function calls to `allocTuple` and `allocTag` respectively. You'll need to implement these two functions in the runtime system.
- A case expression: $\mathsf{case}\ x\ \mathsf{of}\ \overrightarrow{\mathsf{in}i_j\ x_j \Rightarrow b_j}$ can be encoded as an switch structure. The variables x_j should be initialized (by what value?) before the switch structure.

To support the output code, we need to provide a runtime system, including the definition of function `allocTuple` and `allocTag`. Therefore, we need to decide the memory map for these structures.

As we rely on garbage collection to retrieve memory space, some additional information needs to be attached, such as the length of a tuple, or to distinguish between integers, tuples and tagged values. We will discuss about this with more details in the next lab. For now we just use a simple strategy shown in Figure 8 and 9.

As is shown in Figure 8, a tuple with n components will occupy $n+1$ memory cells. The first cell is to store the length of the tuple, the rest n cells store the component values in order. In this lab, each cell is constraint to be 4 Bytes. In this way, `allocTuple(n)` will allocate a memory space of $4 * (n + 1)$ Bytes and

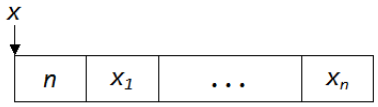

Figure 8: memory map for tuple $x = (x_1, \ ..., \ x_n)$

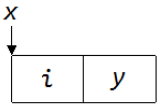

Figure 9: memory map for tagged value $x = \mathsf{in}_i \ y$

store the value n in the first cell. Finally it returns a pointer to the first cell. The value of the components will be loaded later via this pointer.

Note that we will never visit a tuple with the length of 0, so we can simply output "`x=0`" instead of "`x=allocTuple(0)`".

As shown in Figure 9, a tagged value $\mathsf{in}_i \ y$ takes two memory cells. The first cell is to store the tag value i (1 or 2), while the second one for the value to be tagged. In this way, `allocTag(i)` will allocate a memory space of 8kb and store the tag value i in the first cell. And finally it will return a pointer to the first cell. The value y will be loaded later via this pointer.